Martin Korsgaard

Process-Oriented Real-time Programming

Thesis for the degree of Philosophiae Doctor

Trondheim, March 2013

Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Engineering Cybernetics

NTNU – Trondheim
Norwegian University of
Science and Technology

NTNU
Norwegian University of Science and Technology

Thesis for the degree of Philosophiae Doctor

Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Engineering Cybernetics

© Martin Korsgaard

ISBN 978-82-471-4227-1 (printed ver.)
ISBN 978-82-471-4228-8 (electronic ver.)
ISSN 1503-8181

Doctoral theses at NTNU, 2013:65

Printed by NTNU-trykk

Summary

THE CORRECTNESS of a real-time computer system depends, not only on producing correct computational results, but also on the time when those results are produced. Producing a result at the wrong time is considered an error, just as producing the wrong result, or producing no result at all.

Modern programming languages often contain high level features designed to improve the quality of programs, by making them more readable or more maintainable. However, when it comes to control of timing, most languages still include only low-level primitives such as control of thread priorities and access to a clock; these can be hard to use correctly and may lead to programs of low quality. This thesis argues that because timing is no less important than computational results in a real-time system, it should be supported on equal terms with computation by real-time programming languages. Thus the quality standards that apply to the computational primitives of a real-time language should also apply to its temporal primitives.

Real-time programming is an inherently concurrent problem, and to achieve high quality primitives for manipulating timing it is necessary to use an underlying high quality concurrent programming paradigm. One such paradigm is *process-oriented programming*, where systems are constructed using synchronous, message-based communication between self-contained parallel processes. Within this paradigm, no process may affect the state of another, except through mutually agreed upon communication. This increases modularity, making it easier to verify the correctness of a complete system by examining its parts, one at a time. Systems that are more modular are easier to maintain and can be said to have higher quality.

So far, process-oriented programming has not been widely used for real-time applications. One reason for this is that existing process-oriented languages have only limited support for real-time programming. Another reason has been the lack of suitable analysis techniques: while the analysis of timing in traditional real-time systems is a well-developed field, the methods used are often not applicable to process-oriented designs.

This thesis is divided into two main parts. The topic of the first part is the design of real-time language primitives that allow temporal constraints to be implemented while maintaining program quality, and that are compatible with process-oriented design. The topic of the second part is schedulability analysis of process-oriented real-time systems.

The main contributions of the thesis are:

Language Primitives for Process-Oriented Real-time

The thesis evaluates existing real-time language primitives, such as explicit setting of priorities or deadlines, delay-statements and so on, with respect to commonly accepted notions of what constitutes high quality programming. It is argued that several commonly used primitives are of low quality; in particular, the use of scheduling priorities in code is strongly discouraged. It is argued that when implementing a relative time constraint, such as a period or a relative deadline, one should use language primitives that explicitly reference relative times. A new primitive, the "Time-construct" is developed, which assigns a relative deadline to a block of code, while at the same time preventing the block from terminating until its deadline. It is demonstrated that this construct, together with a triggering mechanism for implementing sporadic tasks, allows for intuitive and readable implementations of a wide range of temporal constraints.

Process-oriented design requires the use of synchronous communication rather than mutual exclusion based synchronization, which is currently more widely used. The consequences of using synchronous communication in a real-time setting is discussed and several observations are made regarding how to best implement this type of communication without undermining the ability of the system to meet its temporal constraints.

The Toc Programming Language

A prototype real-time programming language called "Toc" is developed, together with a compiler and a run-time system. Toc is based on the process-oriented language of occam. The language mechanisms in occam that relate to timing are replaced with the primitives that were deemed to have high quality, including the time-construct. The execution model is also replaced: whereas scheduling in occam is largely non-deterministic, scheduling in Toc is based on earliest deadline first (EDF).

It is common for real-time systems to separate between hard and soft real-time requirements, and between real-time and non-real-time components. For the sake of argument this thesis takes the opposite stand; that any part of a real-time system that cannot be given a meaningful deadline for its completion can be omitted entirely. Arguments supporting this hypothesis include (1) if the execution of some component of a program can be postponed indefinitely without violating the system specification, then that component is in fact not required, and (2) if the execution of a program component can not be postponed indefinitely, then it has, by definition, a deadline for its completion.

To test this hypothesis, the Toc scheduler is made *lazy* and will only execute a process if it is explicitly given a deadline, or if its execution is required by a process with a deadline. Processes whose execution is not necessary in order to meet deadlines will never be executed, even when the system is idle. This forces the programmer to become aware of all the deadlines that apply to a system. Moreover, when used in conjunction with the Toc EDF scheduler, no process will be completely denied execution even in times of high loads.

Toc is applied to a case study involving control of an elevator model. The usability of Toc, and the practical consequences of using a lazy scheduler are discussed based on results from that study. It is found that using a lazy scheduler will sometimes require the programmer to specify deadlines that seem obscure or arbitrary, and that are unlikely to be part of the specification of a system; thus at least partially rejecting the stated hypothesis.

Other contributions pertain to schedulability analysis:

Analysis of Synchronous Client–Server Systems.

Systems using synchronous communication have proved to be difficult to analyze with respect to schedulability, limiting their use for hard real-time applications. This thesis develops a schedulability analysis that supports synchronously communicating real-time systems with a client–server structure. The analysis works for uniprocessor systems scheduled using either EDF or fixed priorities. To allow for analysis, a variant of both the priority inheritance and the priority ceiling protocol are developed for these systems.

It is demonstrated that deferring parts of the computation generated by a server call until after the call will in some cases improve schedulability. This transformation is difficult to achieve when using traditional communication methods such as protected objects, because then one cannot move computation out of a critical section while maintaining mutual exclusion.

Current multiprocessor schedulability analyses have only limited support for job-level parallelism (JLP): each real-time task is typically required to be serial, or to have a simple parallel structure. *Malleable jobs*, where the number of processors assigned to a job is dynamic, is not widely supported. This makes it more difficult to analyze process-oriented systems, which often have a complex structure of parallelism. Two contributions improve on this:

Formal Modeling of Processes in Systems with JLP
: A formal model is presented that allows reasoning on the temporal properties of processes with an arbitrary parallel structure. It is defined what it means for a process to be easier to schedule than another, and what it means to have an upper bound on execution time; these new definitions are required when allowing JLP, as execution times can no longer be effectively modeled by scalar numbers. Counterintuitive temporal behavior is demonstrated to be inherent in all systems where processes are allowed an arbitrary parallel structure. For example, there exist processes that are guaranteed to complete on some schedule, but may not complete if executing less than the expected amount of computation. Not all processes exhibit such counterintuitive behavior, and a subset of processes that are well-behaved in this respect is identified.

Fair Intra-Job Scheduling and JLP
: A framework is presented for analyzing complete systems where jobs may have a complex parallel structure. The requirement is that the intra-job scheduler is reasonably fair, that is, it must distribute processors equally to all parallel processes of the same job, even within short windows of time. Communication between jobs is not permitted, though it is possible to model deterministic communication within branches of the same job. Upper bounds on interference and demand are developed. The framework is then used to construct a pessimistic, but sustainable schedulability test for systems scheduled with EDF. The EDF test has poor worst-case performance, but does allow schedulability analysis for a class of systems for which no other analysis currently exists. Moreover, the framework itself should be useful for constructing analyses with better performance.

Contents

List of Figures

Table of Tables

List of Listings

Preface

This thesis presents the results of my doctoral studies with the Department of Engineering Cybernetics (ITK) at the Norwegian University of Science and Technology (NTNU). The project ran between August 2007 and December 2011, and was funded by the department.

Several human beings have helped and contributed in various ways. I would like to thank my supervisor, Associate Professor Sverre Hendseth, for introducing me to field of programming where making things work is considered the easy part, and for stimulating conversations, relevant or otherwise. I would also like to thank my fellow PhD students at the department for our time together and for weekly cake, Christian for all the carrots I stole (sorry), and Kristoffer, my office-mate, for not taking up too much space, and for his relentless support for Ada even in the face of harsh opposition.

And of course my wife Sonja, love of my life, for being there.

Martin Korsgaard
Oslo, October 2012

Chapter 1

Introduction

I identified the machine—it seems to me to be an Old Testament God with a lot of rules, and no mercy.

J. Campbell

To some, computers are a triumph of engineering; immensely complex systems of perfectly interconnected components, with parts that are atomic in scale and astronomical in number. Others find them frustrating, unreliable and short-lived.

When a computer crashes or otherwise fails to do what was expected, this is usually caused by mistakes made by its programmer. A computer will do exactly as told even if given an arbitrarily complex program, but it may not always be clear to the programmer until later what, in effect, the computer was told to do.

The failure of a computer program has the potentially greatest consequences when the computer is used to control or monitor a safety-critical system, such as a control system. Notable incidents caused or amplified by programming errors include the crash of the first Ariane 5 rocket, the Northeast US blackout of 2003, and the out-of-control Therac 25 radiation therapy machine [41, 72, 82].

Control systems have two common properties which complicate their implementation. First, they are real-time, and must be fast enough to keep up with events over which they have limited control. For a real-time system, producing a result too late is an error, no less so than producing the wrong result or no result at all. This is in contrast to non-real-time systems such as a desktop computer, where a late result, although undesirable, may still be of some use. Compare an occasional three second delay in a word-processor to a three second delay by the brake system of your car.

Second, most real-time software is concurrent and consists of multiple threads of execution. Concurrency may add significant complexity to computer programs, in part because of the state explosion problem; it may be intractable for a programmer to anticipate the behavior of a complex concurrent system, even when he knows intimately the behavior of each individual part. A particular difficulty with concurrency errors is that they may only manifest themselves in uncommon situations, and so are less likely to be detected by testing.

Many features of modern programming languages, such as encapsulation and object-orientation, are there to enhance the quality of programs, by making them safer, more readable or more maintainable. Several design philosophies have been proposed in order to achieve the same when programming concurrently. In this thesis, we consider process-oriented programming, a programming paradigm based on synchronous, message-based communication between self-contained parallel processes. Within this paradigm, no process may affect the state of another, except through mutually agreed upon communication. This improves modularity, and helps avoid the race conditions that may occur when concurrent processes compete for reading from or writing to the same memory. The cost is a more rigid structure that also prohibits some legitimate and safe designs.

This thesis argues that because timing is no less important than computational results in real-time programming, it should be supported on equal terms with computation by real-time programming languages.

Unfortunately, the same approach that is used to manage computational results cannot be use to manage timing: whereas programming languages have statements that manipulate the computational result of a program in predictable ways, statements that manipulate timing mostly do so indirectly. A computer cannot be explicitly programmed to complete a task within a certain time limit: If a computer is given a task, it will work on it until the task is completed, or it is interrupted, or until the end of time, with little regard to which it is going to be. The only means of hurrying a task available to the programmer is, if the computer is working on several tasks at once, to prioritize one task over another. Thus managing prioritization of processes often remains the only degree of freedom available for a programmer for ensuring that a real-time system meets the deadlines of its specification.

The thesis will assume a process-oriented design philosophy as a starting point for the concurrency related features of a language; this methodology has already been shown to have many of the desired properties of language constructs, being readable, scalable, safe and maintainable. A valid question is whether it is at all possible to hold a language's real-time features to an equally high standard. For one thing, achieving modularity of program components, so that they can be verified in isolation from the rest of the program, is never entirely possible for timing related properties: whether a component meets its deadline is at minimum dependent on the amount of processing time available, which will always depend on other components in the system. However, it may still be possible to have modularity of implementation, in the sense that the *correctness* of an implementation does not depend on the implementation of other processes. These problems are discussed in part I of the thesis.

Even if a program is perfectly optimized and priorities are optimally managed, a system may still miss deadlines—it may simply not have the computational power required to perform its tasks fast enough. In order to find out whether or not the system will in fact meet its deadlines, one will have to perform schedulability analysis of the system after it has been programmed, taking into account the hardware and environment on which the system is to execute. Alas, it is not possible to accurately predict how much time a computer will require to execute a given program, except in the most trivial cases. Instead, analyses must use pessimistic worst-case estimates, which means that systems may well be deemed unschedulable even when they are not.

Existing schedulability analyses techniques tend not to support the communication primitives or system structures that a process-oriented design requires. Part II of the thesis develops schedulability analyses techniques that allow analysis of certain real-time systems with a process-oriented design.

1.1 Contributions

The thesis includes major contributions in two categories; contributions to the design of real-time language primitives, and contributions to schedulability analysis of process-oriented real-time systems.

The first contributions are related to manners of which a programmer can integrate specification of temporal constraints into a programming language, in ways that are intuitive and that preserve modularity as best as possible.

Contribution 1: The thesis evaluates existing real-time language primitives, such as explicit setting of priorities or deadlines, delay-statements and so on, with respect to commonly accepted notions of what constitutes high quality programming. A new high quality primitive is developed that allows the specification of a wide range of temporal constraints in a readable and modular manner and that is compatible with process-oriented design. The use of synchronous communication in real-time systems is discussed, and a few observations are made as to how this type of communication should and should not be used.

Contribution 2: An experimental programming language called “Toc” is developed that incorporates the new real-time primitive. Toc is lazy scheduled, which means that it does not execute processes without an associated deadline, even if the system is otherwise idle. A compiler and run-time system has been developed. Toc is demonstrated in a case study where a system programmed in Toc is used to control a physical elevator model.

Contribution 3: A uniprocessor schedulability analysis is developed for process-oriented systems where communication has a client–server structure. The analysis supports uniprocessor systems scheduled using either earliest deadline first (EDF) or fixed priorities. To allow for analysis, a variant of both the priority inheritance and the priority ceiling protocol is developed for these systems.

Contribution 4: The thesis presents a formal model for reasoning on the temporal behavior of jobs in real-time multiprocessor systems that allow job-level parallelism (JLP). Counterintuitive temporal behavior is demonstrated to be inherent in all systems where processes are allowed an arbitrary parallel structure. For example, there exist processes that are guaranteed to complete on some schedule, but may not complete if executing less than the expected amount of computation. Not all processes exhibit such counterintuitive behavior, and a subset of processes that are well-behaved in this respect is identified. The analysis assumes that the intra-job scheduler—the scheduler that assigns processors given to the job to the parallel branches of that job—is work-conserving, but otherwise undefined.

Contribution 5: A schedulability analysis is developed for complete systems where jobs are allowed a complex parallel structure. The analysis assumes that the intra-job scheduler is reasonably fair. The analysis does not allow communication between jobs.

1.2 Publications

This thesis is based on the following peer-reviewed conference papers, published between 2008 and 2011:

[61] Martin Korsgaard and Sverre Hendseth. Combining EDF scheduling

with occam using the Toc programming language. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2008*, pages 339–348. IOS Press, September 2008.

[65] Martin Korsgaard and Amund Skavhaug and Sverre Hendseth. Improving real-time software quality by direct specification of timing requirements. In *Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications*, pages 528–536. IEEE Computer Society, 2009.

[62] Martin Korsgaard and Sverre Hendseth. Design patterns for communicating systems with deadline propagation. In Peter H. Welch et. al, editor, *Communicating Process Architectures 2009*, pages 349–361. IOS Press, November 2009.

[63] Martin Korsgaard and Sverre Hendseth. The Computation Time Process Model. In Peter H. Welch et. al, editors, *Communicating Process Architectures 2011*, pages 273–286. IOS Press, June 2011.

[64] Martin Korsgaard and Sverre Hendseth. Schedulability analysis of malleable tasks with arbitrary parallel structure. In *Proceedings of the 17th International Conference on Real-Time Computing Systems and Applications*, pages 3–14. IEEE Computer Society, August 2011.

The first three publications [61, 62, 65] are the basis for chapters 3 and 4, but have been extended and reorganized for inclusion in this thesis. Chapters 7 and 8 are based on [63] and [64], respectively, with minor modifications.

1.3 Thesis Organization

Part I of the thesis discusses real-time programming language primitives with respect to metrics of quality, such as maintainability and readability: Chapter 2 contains background information, including definitions of program quality, and introductions to concurrency, process-oriented programming and scheduling. Chapter 3 evaluates existing real-time language primitives based on the given quality criteria, and suggests a new primitive suitable for programming reactive systems. Chapter 4 describes the experimental programming language Toc, which implements the new primitive.

Part II concerns schedulability analysis: Chapter 5 reviews existing schedulability analysis techniques for both uniprocessor and multiprocessor systems. Chapter 6 develops a novel schedulability analysis for systems that use synchronous client–server based communication. Chapter 7 presents a simple process algebra for reasoning on real-time jobs in systems that allow JLP. Chapter 8 develops a novel schedulability analysis for complete systems that support JLP, when the intra-job scheduler is reasonably fair and when there is no communication between jobs.

Part III includes the thesis conclusion and a list of possible directions for future work.

Finally, the appendixes list the complete syntactic and lexical structure of Toc (appendix A), the full source code of an elevator control system implementation in Toc (appendix B), the nomenclature for part II (appendix C), as well as a list of abbreviations (appendix D) and a glossary (appendix E). The last section in the thesis is the list of references.

Part I

Programming Language Primitives

Chapter 2

Preliminaries on Real-time Programming Languages

British Left Waffles on Falkland Islands

NEWSPAPER HEADLINE

THIS THESIS will discuss programming language primitives for real-time systems, based on the notion that some primitives are better than others even though both can be used to achieve the same effect; that there is a difference between a program being good, and a program that merely works. In order to perform these comparisons with precision it is necessary to first develop some consistent notion of what constitutes quality in programming. This is the topic of section 2.1. An introduction to parallel programming is given in section 2.2, where various concurrency mechanisms are discussed from a program quality point of view. This thesis will focus on the process-oriented programming paradigm, which is introduced in section 2.3. The chapter is concluded by an introduction to real-time scheduling, in section 2.4.

2.1 Quality in Programming

Given two programs that are functionally identical, it may still be that one program is better than another, given some definition of code quality. Likewise, even though two programming languages have the same expressive power, one may be better suited for a certain task, or even better altogether. In this thesis, various real-time programming frameworks will be discussed. These are largely equivalent with respect to expressive power, in that most systems can be programmed within any framework. Still, some languages are more equivalent than others, and to be able to discuss these notions in a more precise manner, some definition of program quality will be necessary.

A starting point is Dijkstra, who argued that the fundamental problem of programming is to manage complexity [39]. As a program grows larger, the programmer must take care to reduce this complexity, or it may easily become too high to cope with. Programs with lower complexity are generally more readable and maintainable, and are likely to have fewer bugs [79].

Reducing complexity can be done by dividing the program into layers and modules. Layers provide abstraction by hiding details about a program component that is not necessary to understand its function. Using higher-level languages is one way of providing this kind of abstraction; indeed, using a high-level language at all, instead of just machine code, was the first and arguably most important step [22].

The other main way of reducing complexity is to partition the program into modules, which allows reasoning on the correctness of a system by looking at one part at a time. The greatest reduction in complexity is achieved when the modules have strong cohesion and loose coupling, which means, respectively, that each module has a clearly-defined functional responsibility, and that it does not rely too much on the implementation details of other modules [96].

The concepts of cohesion and coupling can also be applied to the statement level. Structured programming, like using **for**-loops instead of counters and **goto**s, provides another form of layering and yields higher abstraction. A language feature can be said to have weak cohesion if it requires the implementation of what is functionally a single feature to be spread out across the program, or if it combines multiple non-related functionalities into a single statement. A statement may be said to have tight coupling with another statement if their effects depend critically on each other; this becomes more problematic when the statements are located far apart in code.

Related to the concept of coupling is the concept of safety from Hoare [54][1]. A statement is unsafe if it may fail silently, that is, if neither the compiler nor the run-time system will warn about cases where the ordinary meaning of the statement breaks down. Conversely, a statement is safe if it either succeeds or yields an explicit error. Unsafe statements may result in undefined behavior, and be susceptible to machine- or implementation dependent effects. For example, array operations in C are unsafe, because an array index out of bounds will silently return or modify an arbitrary memory location, whereas an array operation in Java is safe, because it will either succeed or explicitly fail.

[1]Hoare called it "security".

The above concepts will be used to compare programs with respect to quality:

Definition 2.1 (Program Quality)
An implementation A of a program component has low quality if any of the following points hold:

1. A is unsafe (safety).
2. A exposes unnecessary details (wrong level of abstraction, readability).
3. A has an ill-defined responsibility, and should have been split up; or A requires the implementation to be spread out (cohesion).
4. A has hidden side-effects or relies on details about remote parts of the program, or on obscure details about the computing platform (coupling).

An implementation B can be said to have higher quality than A if both produce acceptable results, but B scores better on the above points.

2.2 Concurrency

Concurrency, also known as multithreading or parallel computing, is employed for a number of reasons. For one thing, it is required when executing programs on multiprocessor systems, which again are required to increase performance beyond what a single processor can provide, or to reduce energy consumption. On multiprocessor systems it is the system, not the program, that dictates the use of concurrency; this may leave programmers with the non-trivial task of parallelizing a serial program.

On the other hand, it may be that the program at hand is in itself parallel. This is for example the case for real-time systems, where multiple more or less independent processes must be executed simultaneously on the same computing platform, interleaved in such a way that all processes satisfy their timing requirements. Here, the problem is that of transforming a parallel problem into a serial representation. A system that is both real-time and multiprocessor will have similar challenges, as it is usually not possible to assign a single real-time process to each processor; nor would it be particularly efficient.

Concurrent programming is considered difficult in several existing programming languages, because it often results in low quality programs. Consider the following quote cited in the official documentation of Java Swing [97]:

> It is our basic belief that extreme caution is warranted when designing and building multi-threaded applications, particularly those which have a GUI component. Use of threads can be very deceptive. In many cases they appear to greatly simplify programming by allowing design in terms of simple autonomous entities focused on a single task. In fact in some cases they do simplify design and coding. However, in almost all cases they also make debugging, testing, and maintenance vastly more difficult and sometimes impossible. [...]

Concurrency is essential to real-time programming, because the problem to be solved is fundamentally concurrent. To achieve high quality on the temporal primitives of a real-time language it is necessary to have underlying concurrency mechanisms that also have high quality, and where debugging, testing and maintenance is not "difficult and sometimes impossible". This section will present existing approaches to concurrency in other languages, and illustrate why some have high quality and some do not. Process-oriented programming, which has high quality, will be used as a basis for the real-time primitives developed in chapter 3.

This thesis limits its discussion to imperative, general-purpose programming languages. Completely different approaches to real-time programming have also been successful, such as the synchronous languages of Esterel [13] and LUSTRE [32]. These languages do not have the flexibility of imperative languages—in particular they are limited to finite-state programs—but they have more predictable timing and are easier to verify formally. Lustre has been used in avionics and nuclear power plants [50] where the loss of flexibility to verification is easily justified.

Terminology: Threads, tasks and processes

The terms "thread", "task" and "process" are all used in the literature to describe a parallel component of a program. Process-oriented programming derives its terms from the communicating sequential processes (CSP) [56], where the word "process" has the more general meaning of any program component, parallel or not. This thesis will mainly use "process" in the CSP sense, and explicitly use "parallel processes" to emphasize that a set of processes may execute in parallel. The term "task" will be used to describe top-level processes in schedulability analysis, and for some well-defined real-time concepts such as the periodic or sporadic task. The term "thread" will occasionally be used to explicitly denote a thread of execution.

Listing 2.1: *Parallel in occam*

```
PROC Main()
  PAR
    a()
    b()
:
```

Listing 2.2: *Goroutines in Go*

```
func main() {
        go b()
        a()
        runtime.Goexit()
}
```

2.2.1 Programming Parallel Processes

Say the definition of a program specifies that some processes are to be executed in parallel on a uniprocessor system (ie, interleaved). One implementation approach is to split up processes manually, and cycle through them, executing one piece at a time. This is known as the cyclic executive model [eg, 5]. This is a simple method in principle, and requires no support from the run-time system.

The disadvantage is that all processes become tightly coupled, so that adding or modifying a process may require changes to the other processes in the system. For large systems, this maintenance penalty would be prohibitively large. Instead, concurrent programming languages feature some automatic method for achieving the same effect, by using either a built-in scheduler or taking advantage of OS support.

The method for specifying parallelism varies greatly from language to language. A few languages have explicit parallel constructs, such as **PAR** in occam (see example in listing 2.1) or the **go** statement in Go (listing 2.2).

Ada (listing 2.3) has a task construct that can be used to define a process to be executed in parallel with other tasks. A task is declared like a normal variable, and must have its body defined within the same declaration scope. In Java and C#, parallel processes can be started by instantiating objects from special Thread classes.

C has no built-in support for parallel programming, and an external concurrency library is therefore necessary to write parallel programs. Libraries based on the Portable Operating System Interface for Unix (POSIX) is a common choice on Unix-based system.

2.2.2 Shared Memory-Based Communication

Communication between parallel processes is the main challenge of concurrent programming. Processes can communicate either by sharing memory or by passing messages. Shared memory is the more common of the two, and is the basis for concurrency in eg, Ada, C/POSIX and Java.

Listing 2.3: *Tasks in Ada*

```
procedure Main is
    task Task_B;
    task body Task_B is begin
        b();
    end Task_B;
begin
    a();
end Main;
```

In most systems, sharing memory is the concurrency mechanism closest to the underlying computer architecture, and shared memory based communication typically has high performance in systems where processes indeed share physical memory. Sharing a variable in this case is also easy: A shared variable can be accessed directly by any process that has it in scope, or that knows its memory address.

Sections of code that use shared memory are critical sections, and must be protected against simultaneous accesses to avoid race conditions. This is accomplished by using synchronization primitives such as semaphores or mutexes, or higher level mechanisms such as protected objects or monitors. Concurrent Pascal was the first well-known language to use protected shared memory to communicate between parallel processes, and was the first language to support monitors [20, 55]; a protection mechanism now also used by Java and Ada (in Ada it is called a protected object). A monitor consists of private variables and a set of procedures that can operate on those variables. The procedures are guaranteed to be executed under mutual exclusion, and from everywhere else the private variables are inaccessible.

Concurrent Pascal compilers check for and prohibit unsafe sharing of variables at compile-time, thereby removing race conditions caused by concurrent accesses to a variable. However, one of the advantages of shared memory based communication is the possibility of implementing certain high-performance lock-free synchronization patterns [eg, 53, 80, 98], where carefully written code improves performance in a safe way by omitting synchronization. Lock-free synchronization can yield significant and sometimes necessary performance improvements, so a shared-memory implementation that prohibits all unprotected memory access may not be desirable.

For many languages used in real-time systems concurrency is unsafe by default, so that sharing variables safely requires extra code when it should arguably be the other way around [21]. This is true for eg, Ada, C and Java. Because concurrent sharing of variables in these languages is by default unsafe, any statement on a shared variable becomes unsafe. For example, the statement

```
i = i + 1;
```

may or may not increment i if i is a shared variable.

C has become the most popular language for real-time programming even with no built-in concurrency support. Safe sharing of memory in C is only possible through the use of external libraries, which mostly provide low-level primitives, leaving it to the programmer to ensure correct use. Moreover, one of C's greatest strengths—the flexibility of allowing arbitrary pointers—makes it impossible to implement compiler safety checks for the sharing of variables [100]. The flexibility of C does allow some very efficient synchronization methods, such as lock-free queues, but makes the language inherently unsafe. To compensate for this, subsets of C have been developed that attempt to make the language safer by restricting the use of certain language features; an example is MISRA C [52]. It has also been shown that when using shared-memory based concurrency in languages without inherent concurrency support (such as C) it will always be the case that concurrency errors can be introduced by the compiler itself, for example during optimizations [19, 24].

2.2.3 Message Based Communication

The alternative to shared memory-based concurrency is for processes to communicate by passing messages. Message passing can be either asynchronous or synchronous. Asynchronous message passing is used by the "mailbox" approach of eg, Erlang and Concurrent ML. With asynchronous message passing, senders and receivers are decoupled in time; receivers may choose to wait if no messages are available, but senders are never blocked.

The alternative is synchronous message passing, or synchronous communication, as used by Go and occam, and by the task entries of Ada. With synchronous communication, the sender must always wait for the receiver and vice versa, so that each interaction is a rendezvous, ie, a synchronization point between sender and receiver.

The main advantage of strictly using message based communication is that concurrency becomes safe by default; when tasks do not share memory there can be no race conditions due to unsafe sharing of variables. Message based communication also improves modularity. Each process is only involved in communications that it actively participates in, so the state of the variables of a process will only be changed by itself. This results in a WYSIWYG property not present in shared memory based systems [102], and invisible coupling between processes are removed. Also, where monitors and protected objects should not in general be nested [74], no such restrictions apply to message based communication. In this sense, systems that use message based communication scale better than those based on shared memory.

One drawback of using synchronous communication is related to performance. Synchronous communication always involves at least one task switch. In a setting where task switches are expensive this can represent a significant performance penalty. Both occam and Go get around this by using light-weight processes; process creation and switching times in occam are in the magnitude of tens of nanoseconds [104]. However, light-weight processes are generally not preemptible, which may present problems for real-time systems.

2.3 Process-Oriented Programming

Process-oriented programming is a programming paradigm for concurrency. Key elements are the usage of parallel processes to achieve modularity; the use of synchronous communication between parallel processes, and the absolute encapsulation of process state. The result is a programming technique that ensures that all statements are safe, and where any kind of nondeterminism must be specified explicitly as such.

In a strict process-oriented program, the local variables of a process can only be changed by that process: this is a much stronger guarantee than provided by eg, private variables in object-oriented programming (OOP): in OOP threads of execution are not localized, so a call from one object to another may indirectly result in changes in the first object, making code harder to read.

The process-oriented paradigm was developed along with the occam programming language. The initial target of occam was the Transputer, an early microcontroller capable of parallel execution. Each Transputer could execute multiple threads by using the equivalent of a built-in scheduler, but Transputers could also be used in clusters in order to increase total computational power. When programming occam on the Transputer, the programmer could specify two processes to execute in parallel without needing to know if this meant that they would execute on the same or on different Transputers. Moreover, the programmer was encouraged to always consider whether statements could be executed in parallel, by requiring an explicit keyword also for statements that were to be executed in sequence.

Similarly, the programmer could write channel-based communication between parallel processes without needing to know if the communicating processes would execute on the same Transputer or not. If they were, a channel would be compiled to a virtual, shared-memory based channel. If the processes were on different Transputers then the communication would use a physical link over an actual wire. The occam programming language was ideal for the Transputer, because it allowed the parallelism of a program to be decoupled from the parallelism of its hardware.

In contrast to other imperative programming languages, which use sequential procedures or passive objects to achieve modularization, the primary method for writing a modular occam program is to divide it into multiple parallel processes that communicate through explicitly defined channels. Unlike for example Ada, which contains primitives for synchronous communication, but also allows safe and unsafe sharing of variables, the occam compiler will refuse to compile a program that it cannot verify to be safe. It also strictly prohibits any kind of aliasing (ie, simultaneously referring to the same variable by different names). The result is a language that is somewhat cumbersome to use, but which effectively prohibits many kinds of subtle programming errors such as race conditions or aliasing bugs.

The Transputer is no longer in use, but the Transputer/occam combination has direct a successor in the XCore processor family and XC programing language of XMOS. The occam language is still used in academia, and for complex systems simulation [106]. The newest version of the language is occam-π [104]; a much larger language with support for dynamic and reconfigurable process networks. Process-oriented concurrency libraries have been developed for other languages as well, most notably JCSP for Java [102]. It is also possible to write process-oriented program without explicit language support [99].

2.4 Real-Time Scheduling

Whereas the scheduler in a desktop OS may seek to maximize performance as experienced by a user, the goal of the scheduler in a real-time system is to schedule processes so that they meet their deadlines. How to best accomplish this is discussed here; an introduction to schedulability analysis—determining whether processes in fact do meet their deadlines—is given in chapter 5.

The simplest real-time schedulers are priority driven. In these schedulers, processes are assigned priorities, and the scheduler always executes the highest priority process that is ready. Two scheduling algorithms of this type are fixed priority scheduling (FPS) and earliest deadline first (EDF).

The deadline of a process as compared to a global clock is called the absolute deadline, while the deadline relative to the release of the process, ie, the time when the process becomes eligible for execution, is called the relative deadline. Under EDF, the scheduling priority of a process is defined by its absolute deadline, so that an earlier absolute deadline yields a higher priority. EDF is optimal on uniprocessor systems, in the sense that if a system is schedulable, then it is also schedulable under EDF [75]. EDF is a dynamic priority scheme: the actual scheduling priority of processes depend on their release pattern, which is generally not known in advance.

When using FPS, the priority of each process is set explicitly. The optimal priority assignment on uniprocessors is the deadline-monotonic priority ordering (DMPO), where process priorities are ordered by relative deadlines: a shorter relative deadline results in a higher priority [71]. Real-time systems are often modeled as a set of periodic tasks so that each task has a deadline at the end of its period. In this case the equivalent of DMPO is referred to as rate-monotonic priority ordering (RMPO) [75]. No optimal scheduling strategies exist for multiprocessor systems when relative deadlines are not equal to periods or when processes communicate. This is discussed further in section 5.3.

When processes share resources under mutual exclusion, one process may be blocked waiting for another process to release a resource. This can cause a phenomenon known as priority inversion [67]. In its most basic form, a priority inversion is any situation where a high priority process is blocked waiting for a lower priority process to release a resource; thus some degree of priority inversion is unavoidable when synchronization between processes of different priorities is required. The more serious situation is that of an unbounded priority inversion, which occurs if the lower priority process is interrupted by an intermediate priority process before it completes, leaving the higher priority process waiting for a potentially long chain of processes with lower priorities than itself.

The root cause of the problem is that the scheduler does not actively help to execute code necessary to complete its most urgent process. There are several ways to alleviate the problem. One is to use the priority inheritance protocol (PIP) [35], where a lower priority process will inherit the highest priority of the processes that are blocked by it, thus limiting the priority inversion to one level. The PIP suffers from certain problems; in particular it has poor performance with nested critical sections and can be hard to implement correctly, as exemplified by its inventors failing to do so themselves [107].

Another solution is to use a ceiling protocol, such as the priority ceiling protocol (PCP) [92] or the stack resource policy (SRP) [2]. Compared to PIP, these protocols reduce maximum blocking and also prohibit certain types of deadlock. The SRP is more general as it works with both fixed priority and EDF systems.

The SRP uses a total ordering of processes known as preemption levels. If a process *A* has a lower preemption level (or only "level") than *B*, then *A* will never preempt *B*; ie, it is never the case that *A* is allowed to begin execution when *B* is already executing. For FPS, the preemption level ordering is equivalent to the priority ordering. For EDF, the preemption level ordering is equivalent to the reverse ordering of relative deadlines, ie, a shorter relative deadline implies a higher preemption level.

The SRP works as follows: Whenever a task holds a resource, the resource gains the preemption level of the highest level process that may at some point use it (the ceiling of the resource). The scheduler executes the oldest, highest priority process, and does not allow processes to begin execution unless they have a higher level than all resources. This has three significant consequences: mutual exclusion is achieved without explicit blocking, a higher priority process can be blocked by at most one lower priority process, and no process will be blocked after starting execution [2].

2.4.1 Handling of Temporal Errors

A scheduling overload is a situation where the system is unable to complete all processes within their deadlines. EDF and FPS behave differently during scheduling overload. With EDF there may be a domino-effect where many processes miss their deadlines, but all processes will be executed eventually. Furthermore, if the processes are periodic, then the overload will cause their average periods to increase as if multiplied by the same factor [33], which can be considered a form of fairness.

With FPS, if the overload is caused by one process executing for too long, then processes with a higher priority than the process causing the overload are guaranteed not to be affected. However, other processes will loose deadlines arbitrarily, and, in general, predictability during overload is only guaranteed for the highest priority process [31]. Lower priority processes may never be executed if the overload is permanent.

Some languages [eg, 45, 70, 85] allow the programmer to add exceptions or similar mechanisms that react to missed deadlines, so that they can be handled as an error. This helps for situations where a missed deadline requires an explicit action. However, if the scheduling overload is caused by a misbehaving process that is taking up more CPU time than anticipated, then it is not necessarily this process that will miss its deadlines. Explicit handling of missed deadlines is thus only useful to handle the effect of a scheduling overload, not its cause.

To handle the cause of a scheduling overload, it is necessary to stop tasks from executing for too long. This assumes that the system is known to be schedulable as long as each task executes less than some budget of CPU time dedicated to that task. Then, if a task exceeds its budget it can be stopped or demoted to a lower priority; it will likely miss its own deadline, but will not cause other tasks to miss their deadlines, isolating temporal errors in the system. In some cases a task can be replaced with some form of a degraded backup solution that requires less execution time. When this is not possible, or not enough, then the situation cannot be resolved without violating the specification of the system one way or another.

Execution time budgets are supported directly by eg, Ada 2005 [49] and the Real-time Specification for Java (RTSJ) [85].

Chapter 3

Language Primitives for Process-Oriented Real-Time

– One morning I was sitting in front of the cabin smoking some meat, when—
– Smoking some meat?
– Yes, there wasn't a cigar store in the neighborhood.

G. Marx

Existing programming language mechanisms for controlling temporal behavior range from low-level primitives such as delay functions, to higher level structures that express periodicity and relative deadlines. In this chapter, these mechanisms are evaluated from a program quality point-of-view. It is argued that many low-level primitives, such as explicit setting of priorities, delay-statements and so on, do not hold up to commonly accepted standards for what constitutes high quality programming, and that this can only be remedied by using primitives that have a higher level of abstraction.

Concurrency is an essential property of real-time programming, and high quality concurrency mechanisms are needed to ensure high quality of a real-time system. In this thesis, process-oriented programming is chosen as the concurrency paradigm. Existing real-time languages that support high-level timing primitives are not compatible with this paradigm, as they do not support complex parallel structures nor allow for synchronous communication. Conversely, existing process-oriented languages are not suitable for use in real-time systems, as they provide insufficient control over scheduling, and lack the necessary synchronization protocols.

A new timing primitive is developed in this chapter that allows timing requirements to be implemented in a readable and modular manner, and that is compatible with process-oriented design. The use of synchronous communication in a real-time context is then discussed, and observations are made concerning the consequences of using synchronous communication rather than mutual exclusion based synchronization. These observations are required for the next chapter, when a process-oriented real-time language using the new timing primitive is developed.

3.1 Introduction

To control the temporal behavior of a system, a programmer can use interrupts, access to a clock, some means for delaying execution (eg, sleep()), and the ability to manipulate scheduling priorities. Most existing real-time programming languages require the programmer to use these primitives directly. The objective of this chapter is to develop higher level language primitives for implementing temporal constraints. The primitives should have the highest possible quality according to definition 2.1 (page 11), leading to the following design criteria:

1. The primitives should be safe and not behave unexpectedly.
2. They should be concise and have an appropriate level of abstraction, low enough so that the necessary temporal constraints can be implemented, but no lower.
3. They should not require the implementation of a single temporal constraint to be spread out across the program; ideally, there should be a one-to-one correspondence between a requirement in the specification and that requirement in the implementation.
4. Finally, statements using the primitives should be maintainable and not tightly coupled to other parts of the program. If there is a local change in the temporal specification, it should only require a local change in source code.

The implementation of the primitives themselves must use the means available for controlling the timing of a computer system: access to a clock, adding delays and managing scheduling. Process-oriented programming is chosen as the concurrency paradigm; the primitives must therefore be compatible with process-oriented design, and support the complex parallel structures and synchronous communication that a process-oriented system requires.

All systems to be implemented will be assumed to be reactive systems, ie, systems that do not “explicitly reference the time frame of the enclosing environment” (as defined by Burns and Wellings [28]). In reactive systems, all timing requirements are relative to events in the system and never to an absolute clock. It will also be assumed that all timing requirements in the system are predicates that are either satisfied or not; sliding scale objectives such as minimizing jitter or power consumption will not be considered.

3.1.1 Outline

The structure of this chapter is as follows: Section 3.2 discusses the types of temporal constraints possible in reactive systems, and how these can be implemented in imperative programming languages. Section 3.3 discusses real-time primitives in existing languages and to what extent they satisfy the above design criteria. A new set of language primitives are introduced in section 3.4 that can be used to implement a wide range of temporal constraints. Process-oriented programming requires the use of synchronous communication between processes, instead of the mutual exclusion based synchronization typically used in real-time systems. In section 3.5, the consequences of using synchronous communication in a real-time system are discussed.

3.2 Classification of Temporal Constraints

The term “temporal constraint” will be used to denote any part of the specification of a real-time system that refers to time. A temporal constraint could for example be that some response to an event must be executed within a given deadline, that some process should be repeated with a specified period, or that a system should wait some maximum time for an event, before timing out and doing something else.

In reactive systems, all temporal constraints are relative, and can be classified as being either the minimum or the maximum time between either a stimulus (S) or a response (R) [37]. The full table of combinations are listed in table 3.1.

Constraints 1–4 describe requirements that the environment, including users, are expected to satisfy. A specification containing such a constraint will typically specify some action to be taken by the system if the environment does not satisfy the constraint, for example by timing out and performing an alternative procedure.

Table 3.1: *Stimuli–Response Type Temporal Constraints* [37]

	Environmental Constraints	
1.	min. time from S to S.	Range of intervals between external events.
2.	max. time from S to S.	
3.	min. time from R to S.	Reaction time of environment.
4.	max. time from R to S.	
	Delay Constraints	
5.	min. time from S to R.	A minimum response time requirement.
6.	min. time from R to R.	A minimum period requirement.
	Deadline Constraints	
7.	max. time from S to R	A maximum response time requirement.
8.	max. time from R to R	A maximum period requirement.

Constraints 5–8 describe requirements of the system; that it should neither respond too fast nor too slow. The minimum time constraints can be implemented with delays; the maximum time constraints (ie, deadlines) by manipulating scheduling priorities. A program can enforce a delay, and thus guarantee that a minimum time constraint is satisfied, but cannot do the same for a deadline constraint—a system can merely improve the chances that a deadline constraint is met by assigning it a suitable scheduling priority. In some cases it is possible to prove that a deadline is in fact met by performing schedulability analysis.

In Dasarathy [37], a stimulus was considered to be a signal from the user to the system, while a response was a signal from the system to the user. These definitions are appropriate from the point of view that a specification should only define interactions between the system and the environment, and leave other details to be decided by the implementation. However, in order to achieve modularity it is also necessary to be able to describe the temporal behavior of each module separately, not only the behavior of the complete system. Therefore, stimulus and response constraints will be defined as seen from the module for which the specification applies.

Also, in imperative languages there is typically no explicit communication with the environment; an input may be performed by reading from a particular memory address; an output by calling a particular function. A stimulus may in some cases be an explicit trigger, such as an interrupt, or it may be implicit, such as a state change discovered by polling. It is therefore convenient to define stimuli and responses based on temporal scopes, where a temporal scope is defined as a collection of sequential statements with associated temporal constraints. Temporal scopes can be nested or arranged in sequences, and may overlap. Every stimulus and response can then be associated with the beginning or completion of a temporal scope:

Definition 3.1 (Stimulus and Response)
A stimulus is defined as some abstract condition for allowing a temporal scope to begin execution. A response is defined as the completion of a temporal scope.

With this in mind, the temporal constraints 1–8 in table 3.1 can be given the following imperative implementation outlines:

1–4. The implementation of the environmental constraints is a matter of comparing clock times and taking an appropriate action if a stimulus comes too soon, or timing out if it comes too late. This will require some means for suspending the release of a temporal scope, direct or indirect access to a clock, and some mechanism for timing out.

5–6. Minimum response time constraints can be implemented by using some kind of delay mechanism. It is usually only possible to delay the start of a temporal scope; one cannot explicitly delay its completion.

7–8. Maximum response times are deadline constraints; the meeting of these constraints can be facilitated by manipulating scheduling priorities.

Not all combinations of constraints can be implemented. In general, the actual execution time of a block of code is not known in advance, nor is the delay caused by interference from higher priority processes. A constraint that requires the completion of a temporal scope to be precisely timed is therefore not implementable, because it would require a delay which magnitude would only be known later.

3.2.1 Periodic and Sporadic Task Models

Say a control algorithm should ideally perform its measurements periodically with a period of T, and set its control outputs exactly T time units later, while performing its next set of measurements. An implementation of the algorithm must allow some time for performing the measurements, and for computing and setting control outputs. Say that a mathematical assumption behind the control algorithm is that these times are close to zero, and that the implementation must therefore keep the times small.

A problem with this is that it is difficult to determine how small these times need to be; any chosen upper bound for something that is mathematically assumed to be zero is likely to be artificial. Moreover, the periodic controller would require at least three different temporal constraints; a period, plus maximum delays for reading measurements, and for setting control outputs, which would complicate its implementation.

Because of the complexity, both of correctly implementing such constraints, and of determining what those constraints actually are, complex temporal constraints are uncommon in practical applications. Instead, real-time processes such as the above controller are often given simplified timing requirements. The most common simplifications are

1. The implicit deadline periodic task.
2. The constrained deadline periodic task.
3. The sporadic task.

In schedulability analysis, it is typically assumed that systems only consist of these types of processes. These task models are also supported directly in some real-time environments such as the RTSJ.

The first task type is the implicit deadline periodic task, in which an instance of the task is released periodically, and where the deadline of each instance is set to the next release. Such a task is simple to implement, requiring only a delay mechanism to restrict the period and a mechanism to set scheduling priority to manage the deadline. A generalization of the implicit deadline task is the constrained deadline periodic task, in which the relative deadline is allowed to be less than or equal to the period. These tasks have an equally simple implementation, but are more complicated to analyze.

The implicit and constrained deadline periodic tasks are not defined based on any stimuli, only on responses, and therefore only specify response–response constraints (6 and 8 in table 3.1). Each task instance is a temporal scope, and each response is associated with the completion of a task instance.

To compute the permitted minimum and maximum times between responses in a constrained deadline task, first assume that each instance has some unknown minimum execution time $C_{\min}$. The earliest time an instance can complete is then $C_{\min}$ after its release; the latest an instance can complete while satisfying its temporal constraints is at its deadline, D after its release. The greatest time between completion times is found by assuming

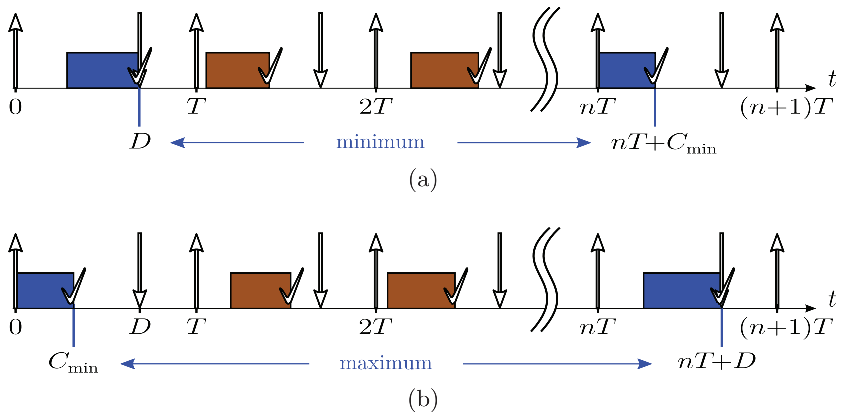


Figure 3.1: Temporal constraints for constrained deadline task: (a) shows minimum response–response time; (b) shows maximum response–response time.

the earliest possible completion of the first instance, followed by the latest possible completion of the last. The least time can be found by assuming the opposite (illustrations are given in figs. 3.1a and 3.1b). From this it can be seen that the time between one response and the nth next response have the following lower and upper bounds:

$$\begin{aligned} \text{min. time from } R_0 \text{ to } R_n &= n \cdot T + C_{\min} - D \\ \text{max. time from } R_0 \text{ to } R_n &= n \cdot T + D - C_{\min} \end{aligned} \tag{3.1}$$

When $C_{\min}$ is not known, a worst-case can be found by assuming it is zero. This yields the following result:

Observation 3.1 (Temporal Constraints of Constrained Deadline Task). *The specification of a constrained deadline periodic task with deadline D and period T is equivalent to the following temporal constraints:*

$$\begin{aligned} \text{min. time from } R_i \text{ to } R_{i+n} &= n \cdot T - D \\ \text{max. time from } R_i \text{ to } R_{i+n} &= n \cdot T + D \end{aligned} \tag{3.2}$$

for all $n \geq 1$, where R_i is the completion of one instance of the task, and R_{i+n} is the completion of the nth consecutive instance after R_i.

In other words, for a constrained deadline periodic task, the maximum response–response jitter for any sequence of instances is $\pm D$. By letting $D = T$, the temporal constraints for an implicit deadline periodic task are obtained:

Observation 3.2 (Temporal Constraints of Implicit Deadline Task). *The specification of an implicit deadline task with period T is equivalent to the following temporal constraints:*

$$\begin{aligned}\text{min. time from } R_i \text{ to } R_{i+n} &= (n-1)\cdot T\\ \text{max. time from } R_i \text{ to } R_{i+n} &= (n+1)\cdot T\end{aligned}\tag{3.3}$$

For an implicit deadline periodic task the maximum jitter is $\pm T$, that is, the completion of two instances can be almost simultaneous, or be up to almost $2T$ apart.

An aperiodic task is defined as a task that is released based on some other criterion than time. Examples include tasks that handle data from an irregular input stream, interrupt handlers, or tasks that handle non-periodic events such as errors. A sporadic task is defined as a repeated, aperiodic task that is also given a minimum inter-arrival time (MIT). A MIT is necessary in order to allow schedulability analysis, else the analysis would need to assume a worst-case where the task executes continuously and inhibits all execution of lower priority processes.

The MIT is an environmental constraint; it cannot be enforced, but violations can be detected. Depending on the application, several choices are available for what to do if the MIT is violated; examples include ignoring excess triggers or storing them in a queue. More complex release criteria can also be implemented, eg, if the sporadic task handles incoming data it can check whether data is available at the end of each instance, and suspend itself only if there is nothing left for it to do.

So long as stimuli do not violate the MIT, a sporadic task definition is equivalent to the following temporal constraints:

Observation 3.3 (Temporal Constraints of Sporadic Task). *The definition of a sporadic task with minimum release time T and relative deadline D is equivalent to the following temporal constraints:*

$$\begin{aligned}\text{min. time from } S_i \text{ to } S_{i+1} &= T\\ \text{max. time from } S_i \text{ to } R_i \quad &= D\end{aligned}\tag{3.4}$$

for all consecutive stimuli S_i and S_{i+1}, where R_i is the response to stimuli S_i.

Although common, approximating complex temporal constraints with periodic tasks may cause unacceptable behavior in some real-time systems. For control systems in particular, low jitter may be essential to control performance [101]; and as illustrated by observation 3.2, an implicit deadline task may exhibit significant jitter without violating its temporal constraints. As shown in observation 3.1, a constrained deadline task can reduce allowed jitter by reducing the relative deadline, but this will also limit the available computation time for the task.

In contrast, by using a repeated sequence of temporal scopes it is possible to specify precisely both jitter and allocated computation time, but the temporal constraints will be harder to meet, and a more complex implementation is required.

3.3 Existing Language Primitives

Some languages, such as the Real-time Specification for Java (RTSJ), support the periodic and sporadic task models directly by allowing periods, deadlines and even MIT violation policies to be specified explicitly when creating a task. A few, mostly academic languages support some form of direct specification of stimulus–response type constraints; examples include Real-time Euclid [60], the Distributed Programming System (DPS) [70], Real-Time C (RTC) [45] and PEARL [46]. Of these, only PEARL has seen widespread industrial use.

In most languages, however, support for implementing temporal constraints consists of lower level primitives such as delay functions, access to clocks, and mechanisms for explicitly setting scheduling priorities. In this section various existing language primitives will be described and evaluated according to the design criteria of section 3.1.

3.3.1 Primitives for Implementing Deadline Constraints

Most computer systems maintain the same speed whether executing an urgent process or not; the only means of hurrying a process is then, if the computer is working on several processes at once, to prioritize one over another. This means that deadline constraints must be implemented using some form of prioritization of processes.

Different types of programming language primitives are available for controlling this prioritization. Four mechanisms will be discussed: relative and absolute prioritization, and the setting of relative and absolute deadlines.

Primitives that Specify Relative Priorities

Traditionally, occam uses a prioritized parallel (**PRI PAR**) to specify process priorities, along with a prioritized alternation (**PRI ALT**) to prioritize communication. These are relative prioritization primitives, as they can only

Listings 3.1 to 3.4: *Uses of the Prioritized Parallel and Alternation*

```
PAR
  PRI PAR
    P
    Q
  PRI PAR
    R
    S

:
```

```
PAR
  PRI PAR
    c ! x
    d ! y
  PRI ALT
    d ? v
      P
    c ? u
      Q
:
```

```
PRI PAR
  P
  Q
:
```

```
PRI PAR
  Q
  P
:
```

specify the priority of a process relative to another process referred to in the same statement. Relative prioritization has multiple issues that complicate the implementation of real-time constraints; in occam-π the **PRI PAR**s were removed, and a statement for setting absolute process priorities was added in their place. However, the mechanism is noteworthy for its unique attempt at process prioritization, and will be discussed here for completeness.

One problem is that relative prioritization allows several types of ambiguity in programs [42]. Combining **PAR** and **PRI PAR**, for example, may lead to parallel processes having incomparable priorities, such as for example P and R in listing 3.1. Moreover, it is not well-defined how to handle combinations of **PRI ALT** with **PRI PAR**, especially since they can be used to specify conflicting requirements: In listing 3.2 the **PRI PAR** prioritizes communication on channel c over communication on channel d, while the **PRI ALT** does the opposite.

A further problem with the prioritized constructs of occam in particular, is that compilers do not have to fully support it. The language reference [91] allows implementations to limit the use of **PRI PAR**, by prohibiting replication or nesting, limiting the number of priority levels recognized, or ignoring it altogether. The Transputer, for example, has only two priority levels, which made it unsuitable for many real-time applications [108].

Furthermore, relative prioritization does not allow composition with respect to correct implementation of deadline constraints. If one process P contains two sub-processes with deadlines 10 and 30, and process Q contains two sub-processes with deadlines 20 and 40, then neither prioritizing P over Q (listing 3.3) nor Q over P (listing 3.4), will yield the desired results of prioritizing processes according to urgency. To achieve proper prioritization in general, all processes must be defined in a single, top-level **PRI PAR**. This means that it is impossible to have a free structure of real-time processes, which is one of the essential requirements for process-oriented design.

Finally, the use of **PRI PAR**s suffers from low readability in that a **PRI PAR** does not reflect the deadlines being implemented; it is impossible to read an ordering of processes under a **PRI PAR** and derive the temporal constraints behind the ordering. This makes the correctness of an implementation hard to verify by examining the source code.

Primitives that Specify Absolute Priorities

A widely available primitive for implementing deadline constraints is a Set_Priority()-type statement that explicitly sets the scheduling priority of the current process. This is used by Ada, C/POSIX, Java, occam-π and others. However, the quality criteria in section 3.1 immediately suggest that scheduling priorities should not be used explicitly in code: Like the prioritized parallel, a Set_Priority() statement does not reflect the timing requirement being implemented; it is essentially unreadable, and its correctness is impossible to verify in isolation. As an example, consider the statement

Set_Priority(5);

The priority, in this case “5”, only indirectly represents a temporal constraint. Its correctness depends on the deadline constraint being implemented, which is not part of the code nor is derivable from it. Its correctness also depends on the priority and deadline of all the other processes in the system, so that the addition or modification of one process may require changes to the priority assignments of other processes. Statements that set priorities are therefore tightly coupled to each other, reducing program quality.

Primitives that Specify Absolute Deadlines

The conversion from deadlines to priorities is one-way only, so in order to achieve close correspondence between the specification and implementation of deadline constraints, there must be explicit references to the deadlines in the source code. Deadline specifications also have the advantage that they are uncoupled and therefore modular; unlike a priority specification, the correctness of a deadline does not depend on other deadlines.

PEARL, as well as some other real-time languages with high-level timing constructs, only support direct specification of priorities, and are therefore unable to provide modularity when implementing deadline constraints.

In contrast, support for EDF was included in Ada from the 2005-standard [30]. The support is somewhat incomplete, and other language features that

Listing 3.5: *Example of* EDF *Support in Ada* [from 27]

```
 1  task A is
 2      pragma Priority(5);
 3      pragma Deadline(10); -- gives an initial relative deadline of 10 milliseconds
 4  end A;
 5  task body A is
 6      Next_Release: Real_Time.Time;
 7  begin
 8      Next_Release := Real_Time.Clock;
 9      loop
10          -- code
11          Next_Release := Next_Release + Real_Time.Milliseconds(10);
12          Delay_Until_And_Set_Deadline(Next_Release, Real_Time.Milliseconds(10));
13      end loop;
14  end A;
```

relate to process priorities, such as inheritance and ordering in queues, do not take deadlines into account [27].

The methods used in Ada to support EDF are limited to low-level primitives for setting the absolute deadline of the current task. In reactive systems, the relative deadlines of the specification must therefore be combined with clock times before being passed on. Creating an implicit deadline task with these methods thus requires the programmer to compute the absolute deadline for each instance, and also to manually insert a suitable delay between the end of one instance and the release of the next. The deadline of the first instance of a process must be set separately at compile-time with a **pragma** directive.

The example in listing 3.5 [from 27] shows an implicit deadline task with a period of 10 ms. The assigned priority (line 2) is the preemption level of the task, which is used for synchronization together with the SRP.

Quality wise there are a number of problems with this approach:

1. The preemption level is set explicitly, and its correctness depends on the relative deadline of all other tasks. This is not an intrinsic problem, however, as the compiler could be made to infer the preemption levels from the deadlines.

2. The solution contains multiple statements (lines 3, 6, 8 and 11–12) for expressing something common to all implicit deadline tasks, indicating that the level of abstraction is too low.

3. There is a lack of cohesion between the statements controlling the period and deadline; their value (ie, "10") must appear in three different

statements.

4. The solution suffers from low readability: the intention of the programmer—to write an implicit deadline periodic task—is not readily apparent from this construction.

A consequence of these quality issues is that the code in listing 3.5 has a subtle programming error. The initial deadline is 10 ms after system initialization, but the Next_Release variable is set to 10 ms after the task is first allowed to execute. This will be later than the initial deadline if the system contains earlier deadline tasks. The result is a more or less arbitrary gap between the first two instances of the task.

Most of the statements in listing 3.5 are used to convert relative deadlines into absolute deadlines, and are required because Ada does not support setting relative deadlines directly.

The use of Ada is not limited to reactive systems and the Ada primitives must allow implementation of both absolute and relative deadline constraints. While relative constraints can be implemented using primitives referring to absolute time, the opposite is not true. Also, it is possible in Ada to wrap uses of low-level primitives into an object with a higher level of abstraction (eg, an implicit deadline task class), which can take care of the details and be easy to use. However, correct implementation of such a class is not trivial, as shown illustrated by the subtle bug in listing 3.5.

Primitives that Specify Relative Deadlines

The above discussion leaves the specification of relative deadlines as the preferred way to implement deadline constraints in reactive systems. The DPS and RTC are two languages that allow this.

The DPS contains language constructs for explicitly implementing temporal scopes. Each temporal scope is given a mandatory delay part, plus an optional execution time part and deadline part. For example, implementing a relative deadline of 20 s for a body of code is written

```
start now within 20 sec do
    /* body */
end
```

It is also possible to specify a form of exception handling for when a deadline is missed. Similarly, in RTC, relative deadlines can be implemented by using a **within deadline** construct. Here, a deadline miss can be handled by using an optional **else** clause, as in

Listing 3.6: *Incorrect Implementation of Periodic Task in* C/POSIX

```
unsigned next = clock();
while (1) {
    P();
    next += 1000000; // microseconds
    usleep(next − clock());
}
```

```
within deadline(60) {
    /* body */
} else {
    /* handle missed deadline */
}
```

These methods for implementing deadline constraints are readable and provide one-to-one correspondence between the specification and the implementation. They also allow deadlines to be specified locally.

Direct specification of relative deadlines is also possible in the RTSJ, as part of a real-time task's PeriodicParameters. However, this does not affect scheduling priority, and is only used for detecting missed deadlines.

3.3.2 Primitives for Implementing Delay Constraints

Consider the problem of assigning a one second period to a periodic task using a relative delay function. A very naive implementation would be to insert a one second delay at the end of each instance; this ensures that the task will execute at most every second, but its period is likely to be longer than that, because the delay does not take into account the execution time of the task itself. Each release would be further delayed compared to a task with a perfect one second period, and the task is said to experience drift.

A more complex, but still naive, implementation is to compute the next point of release, and delay for the remaining time difference using a sleep()-like mechanism to produce a relative delay. An example is shown in listing 3.6.

This implementation suffers from a race condition, which will be referred to as the clock–delay race: The problem is that time will elapse between the reading of the clock and the call to usleep(), so although the task will not experience drift, every release will be delayed for too long, reducing the execution time available for the task (this phenomenon is sometimes referred to as local drift). The effect of this delay is greatly increased if the process is preempted by a higher priority process in the critical section between reading

Listing 3.7: *Correct Implementation of Periodic Task in* C/POSIX

```
struct timespec next;
clock_gettime(CLOCK_REALTIME, &next);
while (1) {
    P();
    next.tv_sec += 1; // 1 second
    clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &next, NULL);
}
```

the clock and delaying. The normal method for ensuring that a process is not preempted—making it non-preemptible—cannot be used, because the critical section ends in a delay: If the process is non-preemptible while it sleeps then other processes are prevented from executing, but if it makes itself preemptible before the delay then the race condition reappears.

One way to avoid these problems is to use a delay primitive capable of delaying until an absolute clock time, such as **delay until** in Ada or clock_nanosleep() in C/POSIX. An example of using the latter is given in listing 3.7.

In reactive systems, delays are always relative to a stimulus or a response. The need to use absolute times to implement these delays is therefore unfortunate; in listing 3.7 this complicates what would otherwise have been a trivial one-liner. This poses a problem for designing a high quality delay primitive for reactive systems: Relative delay statements cannot be used directly because of the clock–delay race, but absolute delays are undesirable, because they require extra code to use, and, in reactive systems, are never directly required.

A solution is to have the delay statement specify delays relative to some given point in code, rather than being relative to the beginning of the delay statement itself. For example, in the RTSJ, the period of a task can be set explicitly, and the waitForNextPeriod() function can be called to delay until the next periodic release. This function will also update the relative deadline and other properties that change with each release.

Other high-level primitives for delays and periods include the **start after** and **every** statements of the DPS; here, a periodic task with a period of 10 s can be written

```
from every 10 sec within 10 sec do
    /* body */
end
```

The same periodic task can be written in RTC using an **every** block:

```
every (10) {
    within deadline (10) {
        /* code */
} }
```

These approaches are much simpler than the equivalent Ada implementation in listing 3.5, largely because the latter needs to compute absolute deadlines and delays from a relative specification.

A problem with the above construct in RTC, however, is that it does not guarantee that the **every** statement and the **within deadline** statements are synchronized, which could increase local drift.

3.3.3 Primitives for Implementing Environmental Constraints

The environmental constraints are those relating to the timing of stimuli from the environment. Implementation of these constraints consists of timing these stimuli, and taking appropriate actions if a stimulus comes too early or too late. Such stimuli will be referred to as *events*, and a language primitive that implements this by explicitly handling an event or timing out will be called a handle–timeout construct.

When handling an environmental constraint there is typically no synchronization between the process raising the event and the handler. It is therefore necessary to take into account that events may be raised when the handler is not ready. Likewise, a timeout is essentially an explicit race condition, and one must decide what to do with events that are raised after the timeout expired.

- One choice of action is to discard all events that are raised when the handler is not ready to accept them. This can be implemented using for example suspend–resume, with the handler calling suspend() on itself and the triggering processes calling resume() on the handler. Note that this mechanism involves another race condition: If the handler becomes ready just before the event is raised then the event will be handled; if it becomes ready just after then the event is discarded.

- Another choice is to use a binary semaphore-style mechanism to implement the event handler. This allows up to one raise to be remembered when the handler is not ready. Using a binary semaphore prevents any single event from being discarded due to a race, but there is still a race condition as to whether two consecutive events will be handled once or twice.

Listing 3.8: *A Handle–Timeout Construct in the DPS*

```
accept on (port1, port2) within 10 sec
    when port1 (arg) : /* statements */
    when port2 (arg) : /* statements */
    when timeout : /* statements to handle timeout */
end accept
```

Listing 3.9: *A Handle–Timeout Construct in Ada*

```
select
   accept Trigger do
      A;
   end Trigger;
or
   delay 1.0;
      B;
end select;
```

- It is also possible to use a counting semaphore-style mechanism, which will avoid race conditions altogether by always providing one handling of the event per raise. However, this is not appropriate for all types of events; examples include events that cause the system to wake up from a sleep state.

A handle–timeout construct is supported directly by the DPS; the syntax is illustrated in listing 3.8. In C/POSIX, semaphores can be used, by using sem_wait() to wait for an event, or its timed counterpart, sem_timedwait() to combine an event handler with a timeout. Signaling the event with sem_post() is then used to raise the event.

A handle–timeout construct can be used to implement both timeout constraints and minimum delay constraints. For example, the Ada construct in listing 3.9 will call *A* if Trigger is called before 1.0 s has elapsed; otherwise *B* will be called. If implementing a timeout constraint, then *A* is called if the constraint is satisfied and *B* otherwise; if implementing a minimum delay constraint, then *B* is called if the constraint is satisfied and *A* otherwise. By nesting the construct it is also possible to implement both types of constraints for a single stimulus.

In the RTSJ, there are no dedicated handle–timeout constructs. Instead, an independent timer such as a OneShotTimer can be used to respond to the timeout, while a separate thread waits for the event. The lack of direct handle–timeout support is somewhat mitigated by having explicit support for sporadic tasks; the RTSJ allows both a MIT and a MIT violation policy to be set directly. The possible MIT violation policies are IGNORE, which ignores violating triggers; EXCEPT, which does the same, but also raises an exception in the thread firing the trigger; REPLACE, which merely changes the deadline of the current task instance; and SAVE, which creates a queue of triggers to be released successively.

The handle–timeout constructs are high quality language primitives and satisfy the design criteria in section 3.1: They are readable and concise, and allow timeout constraints to be implemented directly and to be localized in code. Moreover, by integrating the primitive for handling the event with the timeout primitive, the language (or library) can ensure that the timeout is atomic and therefore safe; a manual implementation would need to take care in order to handle an event and a timeout that happen close in time.

3.4 New Language Primitive: The Time-Construct

In section 3.3 it was argued that most existing primitives for specifying deadline and delay constraints have too low level of abstraction, resulting in implementations of low quality. It was also argued that this is not true for the handle–timeout construct—a common primitive for specifying environmental constraints.

Existing languages that do support high-level primitives for specifying deadline and delay constraints, such as the DPS and RTC, have limitations that make them unsuitable for process-oriented programming as these languages support neither synchronous communication nor an arbitrary structure of parallel processes. A new primitive for implementing deadline and delay constraints will therefore be developed that supersedes these existing high-level primitives. It will be explained how this primitive can be used, together with a handle–timeout construct, to create high quality implementations of periodic and sporadic tasks plus a wide range of stimulus–response type constraints, in a manner that is compatible with process-oriented design.

3.4.1 The Time-Construct

The reasoning behind the new primitive follows an observation about the relationship between the deadline of a process, and the release of a process following in sequence:

Observation 3.4 (Duality of Deadline and Delay Constraints). *Say process B is executed in sequence after process A. If B must be guaranteed a release* *before time d, then the deadline of A cannot be later than d. Conversely, if d is the deadline of A, then B cannot be guaranteed a release before d.*

Moreover, if d is in fact the absolute deadline of A, then there cannot be any temporal constraints that require B to be released before d. As a consequence, if A completes earlier than d, the termination of A, and therefore the release of B, can be delayed until d without violating any temporal constraints.

Now, consider the case where B should not be released until some time r later than d. This can be implemented by inserting an empty pseudo-process C between A and B, and setting deadline of C and the release of B to r; again, the deadline of a process can be set to the same time as the release of a process following in sequence. This is the basis for the time-construct:

Definition 3.2 (The Time-Construct)
The time-construct takes a process P and assigns it a time t, both as minimum response time, and as maximum desired response time.

Each time-construct thus defines a temporal scope. The maximum response time property of the construct sets a relative deadline. It is specified only as "desired" to account for the possibility that the actual response time exceeds t, which may happen during scheduling overload, or due to inconsistent temporal specifications. The minimum response time property sets a minimum time from release to when the construct is allowed to terminate; in practice this is the earliest release time of processes following in sequence. In contrast to the maximum response time, the minimum response time property can be enforced by adding a sufficiently long delay.

The time-construct can be used to create deadlines, periods, delays, sporadic and periodic tasks, and a wide range of other temporal scope specifications. The idea of the time-construct is language-independent: the examples in this chapter will use a C-like syntax; in chapter 4, the construct is given an occam-like syntax.

3.4.2 Required Properties of an Implementation

For the time-construct to work as intended, the language and run-time system must behave in certain ways. An important element is how the base time of the construct is defined: the time which is used to compute the absolute deadline of a construct from its relative time parameter. The basic principle is that the base time should be the deadline of the preceding

Table 3.2: *Use of the Time-construct*

	Description	Code

1	Set deadline *d* to *P*. The block may not terminate before its deadline.	**time** (d) {
		    P();
		}

2	Delay for *t*.	**time** (t) {
		}

3	Implicit deadline periodic task executing *P* with deadline and period equal to *t*.	**while** (true) {
		    time (t) {
		        P();
		    }
		}

4	Constrained deadline periodic task executing *P*, with relative deadline *d* and period *t*.	**while** (true) {
		    time (t) {
		        time (d) {
		            P();
		        }
		    }
		}

5	Sporadic task executing *P*, with a relative deadline and MIT of *t*, released once for each signal of the semaphore sem.	**while** (true) {
		    sem_wait(&sem);
		    time (t) {
		        P();
		    }
		}

time-construct, or if there was no immediately preceding time-construct, the completion time of the previous statement. In particular, the following properties are required by an implementation:

Property 3.1: If two time-constructs are arranged in a sequence then the latter should have its base time set to the deadline of the first. Loops count as sequences.

Property 3.2: If two time-constructs are separated only by flow control mechanisms (eg, **if**, **while**) then they should count as being in sequence.

Property 3.3: If the body of a time-construct begins with another time-construct, then the two constructs should have the same base time.

Property 3.4: After a time-construct terminates, determining the deadline of any time-constructs that follow in sequence, or count as following in sequence, must be considered urgent.

Properties 3.1 and 3.2 eliminates drift and improves readability: A time-construct in a loop will then implement a periodic task without drift, and sequences of time constructs will have a final deadline and combined minimum response time exactly equal to the sum of the times of the constructs.

Property 3.2 is also necessary to allow conditional time-constructs and to ensure that a loop with a time-construct behaves like a sequence of constructs, even when it requires evaluation of a loop condition. A consequence of this rule is that the evaluation of expressions used in control structures will fall under the deadline of the succeeding construct.

Property 3.3 allows a relative deadline and a minimum response time to be set separately, but relative to the same base time, and property 3.4 is needed to ensure that the system is aware of all its pending deadlines.

3.4.3 Implementing Periodic and Sporadic Tasks

The properties of the time-construct makes it suited for creating periodic and sporadic tasks. Because a deadline and minimum termination time is set simultaneously, an implicit deadline task requires just one time-construct and a loop (row 3 in table 3.2). The implementation is quite readable: An implicit deadline task is a loop where each iteration takes t units of time.

If two time-constructs are nested then the innermost process will be assigned two deadlines and two earliest termination times, and must satisfy both sets of constraints. Meeting both deadlines is implied by meeting the earlier of the two deadlines, and not terminating until any of the earliest termination times is implied by not terminating until the latest earliest termination time. A constrained deadline task can therefore be written by

Listing 3.10: *Low-Jitter Controller using the Time-Construct*

```
while (true) {
    time (1) {
        /* Take measurements */
    }
    time (98) {
        /* Compute output */
    }
    time (1) {
        /* Set output */
    }
}
```

nesting two time-constructs in a loop (row 4); the innermost construct sets the deadline and the outermost sets the period. Arbitrary deadline tasks (ie, allowing $d > t$) are not as easily constructed using time-constructs; the behavior if the inner construct has a larger time value than the outer construct will be the same as if the two constructs were swapped (although in the first case it will register as a deadline miss for the outer construct).

A sporadic task is written by having a time-construct following an event handling mechanism (row 5). In this example, the task has a deadline and a MIT of t; a deadline shorter than the MIT can be implemented by adding a nested time-construct.

The implementation of tasks in the manner shown relies on the system having all of properties 3.1 to 3.4. Property 3.1 is needed to avoid drift between instances; the base time of the next iteration must be set exactly to the deadline of the preceding instance. Property 3.2 is needed because there is a loop condition; here it is just "true", but it could be of arbitrary complexity. Property 3.3 is required for the constrained deadline task, to ensure that the deadline counts from the time of the release of each instance. Finally, property 3.4 is required so that the scheduler is made aware of the deadline of the next instance as soon as it is ready; it may be the earliest in the system.

3.4.4 Implementing Stimulus–Response Type Constraints

More complex temporal constraints can also be implemented using the time-construct. For example, consider the control loop illustrated in listing 3.10. The loop has a period of 100 time units, with one unit allocated to taking measurements and one to setting the outputs, while the remaining 98 is allocated to computation. The period here is not set explicitly, but can be read

Listing 3.11: *Combining Handle–Timeout and Time-Constructs*

```
while (true) {
    struct timespec timeout = { 1, 0 }; /* one second */
    if (!sem_timedwait(&sem, &timeout)) {
        time (100) {
            time (10) { /* event received, do something normal */
                P();
            }
        }
    } else {
        time (2) { /* timed out, do something urgent */
            Q();
        }
    }
}
```

by taking the sum of the individual time-constructs. The implementation in listing 3.10 is as concise as is possible; every time-construct corresponds to exactly one temporal constraint.

An example of implementing an environmental constraint is given in listing 3.11. Here, a sporadic task is triggered by a semaphore with a timeout of one second. If a trigger is handled before the timeout then P is executed with a deadline of 10, and a minimum of 100 time units is required until next release. On timeout Q is executed with a deadline of 2, and the next release can be accepted immediately after the deadline. Again, note that each time value in the specification corresponds to exactly one statement in the implementation.

3.5 Synchronous Communication in Real-Time Systems

Communication between processes will typically require some form of synchronization; this may in turn disrupt their temporal behavior and reduce their ability to satisfy temporal constraints.

In shared-memory based systems, synchronization between processes occur whenever one process attempts to access a resource currently held under mutual exclusion by another process. Synchronization protocols (eg, the PIP, PCP or SRP) are used to manage these accesses in order to minimize the effect they have on whether processes meet their deadlines, by transferring some notion of urgency from the process being blocked to the process causing the blocking.

Process-oriented systems do not traditionally associate notions of urgency with communication. The prioritized alternation, for example, cannot be used to take the urgency of the sender into account, only the preferences of the alternation process. Some work has been done on the theory of message-based priority in CSP [68], but this has not resulted in alternative process-oriented programming techniques or language primitives.

In real-time programming, managing the temporal aspects of communication is essential, in order for the implementation of real-time requirements not to be undermined by the side effects of communication. Synchronous communication, as used in process-oriented programming, has different effects on the timing of the processes involved than communication by sharing memory, and will require a different kind of run-time support in order to work well in a real-time setting. Several guidelines have been set out for how to best use shared memory-based communication in a real-time setting (examples include keeping critical sections short, and not nesting them when using the PIP [eg, 107]). In this section, similar guidelines are set out for synchronous communication.

3.5.1 Synchronization Protocols for Synchronous Communication

Unless some notion of urgency is associated with the need to communicate then priority inversions will quickly ensue; while process A is waiting to communicate with process B, then B must be considered at least as urgent as A.

When scheduling synchronous entry calls in Ada, the process with the entry will have its priority raised to the highest of any of the processes that wait for it, thus limiting priority inversions to one level. However, the mechanism only works one way: if the process serving the entry has a deadline for itself, then it cannot raise the priority of the process it is waiting for because it does not know which it is—any number of processes may make calls to the same entry, and as callers do not commit in advance there is no method for determining who the next caller will be. For a synchronization protocol to be effective, an urgent process must always be able to identify the processes that are blocking it. If this process can be identified, and its relative urgency increased, then unbounded priority inversions can be avoided:

Observation 3.5. *Sufficient and necessary conditions for developing a synchronization protocol is that for each urgent process that is blocked, it must be possible both to identify which process that it is waiting for, and to promote the execution of this process, eg, by raising its scheduling priority.*

Two synchronization protocols for the special case of client–server systems are developed in section 6.3. Here, client processes may have deadlines, but when a client wishes to communicate, it must be by performing a call to a specific server process. Server processes may accept calls from multiple clients without knowing which will perform a call first, but are not allowed to have deadlines. Therefore, it is never necessary to increase the urgency of a potentially undetermined client in order to meet the deadline of a server, and the requirement in observation 3.5 is satisfied.

In occam, channels are one-to-one, so a synchronization protocol can be implemented that will work with channels and that is symmetric (one is developed in section 4.4.4); though a related problem occurs if there is an explicit deadline for completing an alternation. Therefore, in general,

Observation 3.6. *Processes that communicate with an unspecified client, such as occam **ALT** or Ada **select**, should not be subject to deadline constraints of their own.*

The restriction only applies to programs where it is urgent that some alternating process communicates, but it is deliberately specified as nondeterministic which process that it should communicate with. Such a construction is not likely to be part of a real-life program, so the restriction in observation 3.6 is not likely to be problematic in practice.

3.5.2 Inconsistent Specifications

When communication is synchronous it is not difficult to construct programs where communication between processes makes it impossible for them to satisfy their temporal constraints, no matter which synchronization protocol is applied. For instance, if A wants to communicate synchronously with B, but only after at least 10 seconds, and B wants to communicate synchronously with A, but not later than in 5 seconds, then the specifications are inconsistent and cannot both be satisfied. If using the time-construct, then the deadline constraint will yield for the delay, and B will be said to have stalled.

In general, a process with a deadline constraint should therefore never attempt to synchronize with a process that may be subject to a delay constraint. Although such an arrangement could be made to work if the processes were both written with this in mind, the resulting code would nevertheless have low quality, as changes in the temporal specification of one process could cause unexpected problems in the other.

Observation 3.7. *To avoid stalls, a process subject to a deadline constraint should not communicate synchronously with a process that may be subject to a delay constraint.*

Observation 3.7 prohibits direct synchronous communication between processes that contain time-constructs. In process-oriented systems, such processes should instead communicate via an intermediate process, which could serve as a form of temporal buffer between the processes—though this will only work if the purpose of the communication is the exchange of data and not the synchronization in itself.

Raising an event is a type of communication where the source has a deadline constraint, and where the target in many cases has a delay constraint (eg, a MIT). The purpose of the communication, as seen from the source, is to mark a point in time that may be used as a reference for temporal constraints belonging to the target process, and thus temporal buffering is not permissible. A consequence of observation 3.7 is therefore the following:

Observation 3.8. *Mechanisms for raising events or triggering sporadic tasks should not be synchronous.*

As an example, consider the handle–timeout construct in listing 3.9: Unless the triggering mechanism is asynchronous, the process raising the event will be blocked when the handler is not ready or has timed out. This is usually not desirable; even less so if the event is raised from performance critical code such as an interrupt. In Ada, a non-blocking raise can be implemented in the triggering process by using a conditional entry call rather than committing to a regular entry call. Then, the event will not be raised unless the handler is ready, resulting in a non-blocking suspend–resume style triggering mechanism.

A similar restriction is implied by observation 3.5. Say a process wishes to communicate with an event handling process, but not in order to raise its event. If the handling process is currently waiting for an event then there is no way to hurry its execution. Therefore, in order to enable the use of synchronization protocols, no process should take the initiative to communicate synchronously with an event handling process.

3.5.3 Deadlock

Process-oriented programs often conform to a design pattern that helps guarantee the absence of deadlocks. One is the client–server paradigm [78, 105]. Here, each process acts as either a client or a server in each communication, and each communication is initiated by a request and terminated by a reply.

Theorem 3.1 (Welch et al. [105]). *If the following three criteria are met, then the system is deadlock-free:*

1. *Between a request to a server and the corresponding reply, a client may not communicate with any other process.*
2. *Between accepting a request from a client and the corresponding reply, a server may not accept requests from other clients, but may act as a client to other servers.*
3. *The client–server relation graph must be acyclic.*

An advantage of this method is that it works well with the requirement in observation 3.5 and is therefore well suited to use with a synchronization protocol.

An alternative design pattern for deadlock-freedom is IO-PAR [105], where absence of deadlock is guaranteed when all processes proceed in a sequential cycle of computation followed by all IO in parallel. The whole system must progress in this fashion, similar to bulk synchronous parallelism (BSP) [93]. This pattern is not so suitable for real-time systems, because it implies that synchronization must always involve all processes, inhibiting the specification of local delay and deadline constraints.

It is also possible to use a model checker such as SPIN [57] or FDR2 [43] to prove the absence of deadlocks under more general conditions.

3.6 Discussion

At the beginning of this chapter, a list of four criteria for a high quality real-time language primitive was given, derived from the programming quality criteria in definition 2.1. The primitive should (1) be safe and behave intuitively, (2) be concise and at an appropriate level of abstraction, (3) allow one-to-one correspondence between a requirement in the specification and that requirement in the implementation, and (4) be maintainable and not tightly coupled to other parts of the program.

Existing primitives for implementing temporal constraints were evaluated with respect to these criteria. It was found that in general, using primitives at a low level of abstraction leads to low quality programs, and moreover, that a real-time language targeting reactive systems should have primitives that allow explicit specifications of relative temporal constraints, rather than using primitives based on absolute time, as this will greatly increase readability.

The time-construct satisfies these criteria. It is safe, in that it has two basic functions which it performs: it sets the deadline and minimum termination time, with the minimum termination time always being maintained, and the deadline being maintained if possible. It allows concise and readable implementations of temporal constraints for reactive systems; typically with one statement per timing requirement. Also, the deadlines set by a time-construct are not coupled to other constructs, which helps maintain modularity of temporal implementations.

It may be argued that the setting of deadlines and termination times represents two functionalities that should not be made into one statement, and that time-constructs therefore have weak cohesion. The design of the construct was motivated by observation 3.4, noting the duality between the deadline constraint of one process and the release of the process following in sequence. This duality is what allows the simple implementation of eg, the implicit deadline task and the controller in listing 3.10. Also it allows an intuitively simple reading of code using the time-construct: A time-construct with time t will require t time to execute. Nevertheless, as a result of this dual functionality there are certain sets of temporal constraints cannot be implemented using time-constructs, such as periodic tasks where $D > T$.

Several observations were made concerning the use of synchronous communication in a real-time setting. Observation 3.5 describes a necessary condition for developing a synchronization protocol: whenever an urgent process is blocked it must be possible to identify which process it is waiting for and to promote the execution of that process. Observation 3.6 notes that this condition is not satisfied for entry calls or alternations whenever the urgency of the alternating process is higher than that of any potential clients. This implies that some restrictions on process organization are necessary in order to apply synchronization protocols to synchronous communication.

The simultaneous setting of deadlines and termination times has an impact on which processes that can communicate without causing stalls. Observation 3.7 implies that when using time-construct to implement temporal constraints, two processes that contain time-constructs should not communicate directly; this is more restrictive than if deadline and delay constraints were specified separately.

Observation 3.8 stated that mechanisms for raising events or triggering sporadic tasks should not be synchronous, as it may cause the firing process to stall. This necessitates the inclusion of at least one non-synchronous mode of communication even in a process-oriented system.

Finally, deadlock avoidance in synchronously communicating systems was discussed, and it was concluded that one technique for avoiding deadlocks, the client–server paradigm, is suitable for real-time systems; while another, IO-PAR, is not. Guaranteeing the absence of deadlock is a necessary condition for schedulability analysis, and the client–server paradigm will be used for this during the analysis in chapter 6.

The next chapter introduces the programming language Toc, which describes an implementation of the time-construct in a process-oriented language. A case study of controlling an elevator is performed; the quality and usability of the time-construct is then discussed based on concrete examples.

Chapter 4

The Toc Programming Language

Wouldn't the sentence "I want to put a hyphen between the words Fish and And and And and Chips in my Fish-And-Chips sign" have been clearer if quotation marks had been placed before Fish, and between Fish and and, and and and And, and And and and, and and and And, and And and and, and and and Chips, as well as after Chips?

M. Gardner

The previous chapter discussed the design of real-time language primitives for process-oriented systems with respect to program quality. It was argued that the specification of explicit temporal scopes using time and handle–timeout constructs would allow a wide range of temporal constraints to be implemented in an intuitive manner. A few observations were also made regarding the use of synchronous communication in real-time systems.

In this chapter, these findings will be used to design a process-oriented real-time programming language, which will be called "Toc". Toc is based on a subset of occam 2.1, with the addition of the new timing primitives, and a new, real-time scheduling model. A prototype compiler and run-time system have been developed for Toc.

4.1 Introduction

The primitives described in the previous chapter were language independent, so in order to experiment with them they had to be integrated into an actual language. Occam 2.1 was chosen as a basis for this language because of its simplicity and consequent ease of implementation. Also, being a process-oriented language, occam is able to realize the high quality implementation of concurrency that was hoped to achieve for the implementation of temporal constraints.

An important part of the design of a real-time, concurrent language is the scheduling model. It was decided to use EDF for scheduling Toc programs, because this was thought to be a better match for the time-constructs, which use deadlines in their specification.

Another scheduling design choice is what to do with a process that is neither subject to temporal constraints of its own, nor needs to be executed in order for other processes to meet their temporal constraints. In fixed priority systems one may set a default priority that applies to such processes. When using EDF it would be possible to assign a default deadline, such as a "very late" deadline, later than any deadlines specified explicitly. The default priority or deadline could then be used for background tasks, so that they are scheduled for execution only when no real-time tasks are eligible for execution.

The Toc scheduler on the other hand, does not set default scheduling parameters this way. Instead, the Toc scheduler is *lazy*, and only executes code that must be executed in order to meet a deadline. Code that is not subject to a deadline constraint will never be executed, even if the system is otherwise idle. This choice is motivated by the following hypothesis:

Hypothesis 4.1 (Laziness Hypothesis). *Any part of a real-time system that cannot be given a deadline for its completion can be omitted entirely.*

The initial arguments in favor of the hypothesis were (1) that if the execution of some component of a program can be postponed indefinitely without violating the system specification, then that component is in fact not required, and the system will satisfy its specification even without this component. (2) Conversely, if the execution of a program component cannot be postponed indefinitely, then it has, by definition, a deadline for its completion.

Experimenting with the validity of the laziness hypothesis and the practical implications of using a lazy scheduled real-time programming language were some of the motivations behind the implementation of Toc.

4.1.1 Outline

This chapter continues with a brief introduction to occam 2.1, on which Toc is based. The details of the Toc language are presented in section 4.3;

the Toc scheduler in section 4.4. In section 4.5, some details about the implementation of the compiler and run-time system are discussed. Section 4.6 presents a case study where Toc is used to control a model elevator. The chapter concludes with a discussion about the usability of Toc and the validity of the laziness hypothesis.

4.2 The occam 2.1 Programming Language

Toc is based on occam 2.1, a concurrent, process-oriented, imperative programming language inspired by CSP. This section provides a brief introduction to the language and its scheduling model.

4.2.1 occam 2.1 Language Fundamentals

Only a very brief presentation to occam will be given here. The reader may refer to the language reference [91] for a more in-depth introduction.

occam does not have quite the same lexical rules as other languages. For example, periods (`.`) are permitted in identifiers, and asterisks (`*`) are used rather than backslashes as escape characters in strings. Also, occam uses layout resolution rather than curly braces to identify blocks of code, as does for example Python and Haskell, but the occam rules are more strict and requires each block level to add exactly two spaces of indentation.

All occam programs are constructed using a set of primitive processes. In occam 2.1 these are assignment (`:=`), channel output (`!`), channel input (`?`), **SKIP**, which does nothing, and **STOP**, which halts the program. There are also primitive processes for reading clocks and for timeouts.

Composite processes are made by using a set of compound constructors. These are **SEQ**, **PAR**, **ALT**, **IF**, **WHILE** and **CASE**. The compound processes **SEQ**, **PAR**, **ALT**, and **IF** may be replicated by using the **FOR** keyword. They may also be nested. Perhaps uniquely to occam, the **SEQ** constructor is needed to make two statements execute in sequence. The constructors **PAR** and **ALT** may also be prioritized, as discussed in section 3.3.

occam contains the typical set of arithmetic, logical and bitwise operators found in eg, C, but unlike C, occam expressions cannot have side effects. Also, there is no operator precedence or associativity, so all expressions with more than one binary operator must be parenthesized.

In occam, sharing a variable between parallel processes is not legal unless it is used read-only by all the parallel components; instead, processes communicate using channels. Channels may be used to exchange primitive data such as integers, but may also have more complex protocols, including sequence protocols and case protocols.

The alternation (**ALT**) is used to explicitly introduce non-determinism. An **ALT** contains a set of pairs of guards and actions: a guard consists of a Boolean expression and/or a channel communication; an action may be any process. The **ALT** will execute one non-deterministically chosen action for which the associated guard is *open*; that is, if it has a boolean expression then this must evaluate to **TRUE**, and if there is a channel, then it must be ready to communicate. In occam 2.1, a communication guard can only be an input process.

Safety in occam 2.1 is ensured by a set of usage rules. One rule states that if a channel is shared by multiple components in a **PAR**, then it may only be used for input by one parallel component, and only for output by one parallel component. Another rule states that a variable that is assigned to by any component in a **PAR** may not be referenced by any other component in that **PAR**. There are several other usage rules, such as a rule prohibiting aliasing (referring to the same variable by different names).

The usage rules are enforced at compile-time. For all usage rules, the elements of an array may be considered separate variables with respect to the rules, but only if it can be determined at compile-time that the usage rules for each individual element are not violated.

An extended rendezvous is a language extension from occam-π that is not part of occam 2.1. It allows the programmer to force the execution of a process "in the middle" of a channel communication; that is, after the processes have rendezvoused but before they are released. The process to execute is called the during-process. Occam-π only supports extended inputs, which are written using the double question-mark operator, as in

```
channel ?? var
   During(var)
```

Extended rendezvous turned out to be a necessary addition to the Toc language (this will be explained in section 4.4.6). For more information about extended rendezvous or other occam-π language extensions, see the occam-π quick reference guide [103].

4.2.2 Occam 2.1 and Fairness

As a concurrent, non-real-time language, occam 2.1 has an execution model based on non-determinism: it is generally not defined which process to execute, or which alternative to take, if multiple choices are available.

In non-real-time systems, programmers tend to expect that the scheduler is reasonably fair and allows each process a more or less equal share of processing time [69], and most non-real-time schedulers maintain at least some degree of fairness. However, because scheduling decisions in non-real-time programs should be considered non-deterministic from a programmer's point of view, the correctness of a well-written program should not depend on the decisions made by the scheduler. If fairness between processes is significant for an implementation to work, then this must be handled explicitly. The following examples illustrate this.

Example 4.2.1 (Fairness in Go)
Consider the Go program in listing 4.1, which prints `"a"`s and `"b"`s in parallel to stdout. An execution of this program (compiler: `6g r60.3 9516`) resulted in the sequence

$$\langle a, b, a, b, a, b, a, b, a, b, a, b, a, b, a, b, a, b, \dots \rangle \tag{4.1}$$

with minor deviations from perfect alternation occurring every few thousand elements. An execution of the same program when compiled with a different compiler (`gccgo 4.6.1`) resulted in the more irregular

$$\langle b, b, b, b, a, b, b, a, a, b, a, b, a, b, a, b, a, b, \dots \rangle \tag{4.2}$$

There is nothing in the program in listing 4.1 that dictates fairness, so none of these outputs can be considered more correct than the other. In fact, an output in which only `"a"`s were printed could also be considered equally correct.

Unlike the internal scheduler in Go, an occam 2.1 scheduler typically makes no attempt at being fair. Occam 2.1 has a prioritized alternation construct, **PRI ALT**, which guarantees priority to textually earlier alternatives, unlike the standard **ALT**, which makes no such guarantees. However, most compilers treat every **ALT** as a **PRI ALT** [69], making **ALT** and **PRI ALT** equally unfair. Similarly, there is a **PRI PAR**, which guarantees priority to textually earlier processes, as opposed to the **PAR** which makes no such guarantees, but again, a programmer cannot rely on a **PAR** to exhibit more fairness than a **PRI PAR**.

Example 4.2.2 (Fairness in Occam)
As an illustration of this, consider the occam 2.1 program in listing 4.2, which also prints `"a"`s and `"b"`s in parallel to stdout. The occam version is more complex than the Go version, mainly because it is not legal in occam for two processes to share stdout, so that a separate process must be made to function as a multiplexer.

Listing 4.1: *Fairness of Go*

```
func printer(s string) {
        for {
                fmt.Println(s)
        }
}
func main() {
        go printer("a")
        go printer("b")
        runtime.Goexit()
}
```

Listing 4.2: *Unfairness of occam*

```
PROC output.multiplexer (CHAN BYTE out, [ ]CHAN BYTE input)
  WHILE TRUE
    BYTE in:
    ALT i = 0 FOR SIZE input
      input[i] ? in
        SEQ
          out ! in
          out ! '*n' -- newline
:
PROC printer (CHAN BYTE channel, VAL BYTE c)
  WHILE TRUE
    channel ! c
:
PROC main (CHAN BYTE stdin?, stdout!, stderr!)
  [2]CHAN BYTE cs:
  PAR
    output.multiplexer(stdout, cs)
    printer(cs[0], 'a')
    printer(cs[1], 'b')
:
```

Executing this program (compiler: `kroc` 1.4.0) results in the sequence

$$\langle b, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, a, \dots \rangle \tag{4.3}$$

After the first `"b"` the program will only print `"a"`s.

The favoring of `"a"`s in example 4.2.2 occurs because the **ALT** is, in effect, implemented as a **PRI ALT**. Being completely unfair may be counterintuitive, but it has the advantage of not even giving the illusion of fairness; an illusion which could conceivably cause failure in systems where fairness is required, and where the scheduler behaves fairly in most, but not all of the time. For example, a programmer that only checks the first few hundred lines of output from eq. (4.1) may be lead to believe that this program is perfectly fair, and will output `"a"`s and `"b"`s in perfect alternation, even though this is not the case.

The choice of being completely unfair has a similar rationale to the laziness hypothesis: a programmer cannot rely on a program exhibiting behavior that is not required of it, and if something is not required, then it may be omitted.

In contrast to occam, Toc is a real-time programming language, so notions of fairness are less relevant. Instead, most scheduling decisions are made based on real-time constraints. However, there may still be situations where real-time constraints are not sufficient to decide which processes that are to be executed, for example when several parallel processes must be executed to meet a single deadline. In these situations, the Toc scheduler will behave like the occam scheduler, and schedule eligible processes without providing any kind of fairness guarantees.

4.3 The Toc Programming Language

Toc is based on a subset of occam 2.1, and—except for some additions and omissions—has the same structure of primitive and compound processes, and requires programs to satisfy the same usage rules. Toc is to be considered an experimental language, and some features of occam 2.1 that did not seem necessary for experimenting with Toc have been omitted. There are also minor syntactic differences, mostly made because they simplified writing the Toc compiler; these are to be considered limitations of the existing version rather than changes made for their own sake, and are listed in section 4.5. The complete syntactic and lexical structure of Toc is listed in appendix A.

Table 4.1: *Use of the* ***TIME*** *Construct*

Use	Code
Set deadline *d* milliseconds to process *P*. The construct may not terminate before its deadline.	**TIME** d **MSEC** P()
Delay for *t* seconds.	**TIME** t **SEC** **SKIP**

Periodic task executing process *P*, with deadline and period equal to 10 µs.	**WHILE TRUE**
	  TIME 10 **USEC**
	    P()

Periodic task executing process *P*, with relative deadline *d* and period *t*, both of **TIMESPEC** types.	**WHILE TRUE**
	  TIME t
	    TIME d
	      P()

4.3.1 The TIME Construct

Toc includes a new type, **TIMESPEC**, which can represent relative or absolute time. The postfix operators **DAY**, **HOUR**, **MIN**, **SEC**, **MSEC**, **USEC** and **NSEC** take an integer value and return a **TIMESPEC**. Plus, minus and relational operators can be used with **TIMESPEC** expressions, and the keyword **NOW** can be assigned to a variable to get the current clock time.

TIME is the Toc implementation of the time-construct. It takes an expression of **TIMESPEC** type and a process, and assigns the specified time both as a relative deadline and a minimum completion time for the process. An example is

```
TIME 10 MSEC
  PRINT "Hello World!"
```

If a deadline is missed, then a run-time warning will be issued, and the base time of any subsequent **TIME** constructs moved forward accordingly.

A few examples are given in table 4.1; these correspond to examples in table 3.2 of the previous chapter.

4.3.2 The HANDLE–TIMEOUT Construct

The other new construct in Toc is the **HANDLE** construct, with an optional **TIMEOUT** part, which implements a handle–timeout construct. Toc contains an event type, **EVENT**, to use with this construct. An event variable counts the number of times it has been raised (with **RAISE**). A **HANDLE** construct takes an event variable and a process, and if the event variable

Table 4.2: *Use of the **HANDLE** Construct*

Use	Code
Sporadic task with MIT and relative deadline *t*, signaled by executing **RAISE** e.	**WHILE TRUE**
	HANDLE e
	TIME t
	P()
If *e* is raised within 1 s, execute *P* with a deadline of 10 ms, and delay 100 ms until next possible handling of *e*. If *e* is not raised before the timeout, execute *Q* instead with a deadline of 2 ms.	**HANDLE**
	e
	TIME 100 **MSEC**
	TIME 10 **MSEC**
	P()
	TIMEOUT 1 **SEC**
	TIME 2 **MSEC**
	Q()

has a positive raise count, it will be decremented, and the process executed. A **HANDLE** construct will block while the raise count is zero.

The first statement in the handle block is typically a **TIME** construct, so that subsequent statements in the block are assigned a deadline.

Variables of the **EVENT** type must satisfy an additional usage rule, stating that if an **EVENT** is shared by multiple components in a **PAR**, it may only be used in a **HANDLE** construct by one parallel component.

The reason not to use synchronous channels to communicate events is that this could lead to stalls as noted in observation 3.8, as a task wishing to raise an event when the handler was not ready would be blocked. Raising an event with the **RAISE** primitive, on the other hand, is an operation which is guaranteed not to block. The **EVENT** type also carries timing information: A **TIME** construct in sequence to the **HANDLE** gets its base time set to either the time of the **RAISE**, or the time of the **HANDLE** becoming ready, whichever is latest.

Two examples of using the **HANDLE** construct are shown in table 4.2. The first is a sporadic task with minimum inter-arrival time (MIT) and relative deadline *t*, released by raising the event *e*. The task will be released one time per raising of the event. The second example shows a process with fairly complex temporal behavior, including a timeout.

If placed in a loop then the body of a **HANDLE** construct will be executed exactly once per raising of the event; that is, the event trigger mechanism is similar to a counting semaphore. The decision to implement it this way, rather than eg, a suspend–resume, was not made because a counting semaphore is intrinsically more useful; on the contrary, events are often used

Listing 4.3: *Binary Semaphore Style Event Manager*

```
PROC Event.manager(EVENT e, CHAN INT raise, ready)
  BOOL handler.ready:
  INT x:
  ALT
    handler.ready & raise ? x
      SEQ
        handler.ready := FALSE
        RAISE e
    NOT handler.ready & raise ? x
      SKIP
    ready ? x
      handler.ready := TRUE
:
```

to wake up processes that have been idle, in which case duplicate signals should be forgotten, and a suspend–resume would have been more practical.

However, when more complex criteria exist for coordinating events, it may be necessary to have an event managing process between the raising processes and the handling process, which can take in event requests and dispatch them as needed. In this setting, what is needed is an event primitive that is suitable as a building block for programming more complex behavior. A suspend–resume mechanism is less appropriate for this; in particular, the race condition between suspend and resume can make their behavior difficult to anticipate. The behavior of a counting semaphore is race-condition free—it always provides one release per raise—and is therefore a better choice for a building block.

As an example, a binary semaphore type event mechanism can be implemented with **EVENT** and **HANDLE** by using an event managing process, as shown in listing 4.3. Here, the process with the **HANDLE** block sends a message to the event manager that it is ready to receive a new event immediately after receiving the previous event. The manager discards all incoming events until it has received such a message. Processes wishing to raise the event must signal the handler using a synchronous channel.

4.4 The Toc Scheduler

This section explains the design and implementation of the lazy EDF scheduler of Toc, an EDF scheduler with the additional property that it does not execute processes without an associated deadline, even if the system is otherwise idle.

4.4.1 Basic Scheduling Algorithm

In section 3.3 the explicit use of priorities to implement deadline constraints was discouraged, because of problems with readability and maintainability. However, this does not prevent the use of priorities for scheduling; a compiler can convert deadlines in code to static priorities using for example DMPO, which would avoid the problems of coupling and readability, while allowing the scheduler to use fixed priorities.

Nevertheless, EDF was chosen as the basic scheduling algorithm for Toc. One reason was that EDF has better performance on uniprocessor systems, because it retains more information: the conversion to fixed priorities is lossy and is unnecessary when the actual deadlines are known.

Another reason is that EDF handles scheduling overloads in a more appropriate manner: When systems scheduled with FPS become overloaded, processes with low priorities may never be able to execute, even though their low priorities are set due to scheduling considerations, not due to a low relative importance. Using FPS therefore assigns a secondary semantic meaning to the setting of a process deadline: any process with a long relative deadline is implicitly considered less important than one with a short deadline. In contrast, using an EDF scheduler will prevent long deadline processes from being completely denied execution.

The Toc scheduler is also lazy. The use of a lazy scheduler is permitted if one accepts the laziness hypothesis (hypothesis 4.1): If a process does not have a deadline, then it can be delayed indefinitely without violating its temporal constraints, and this is precisely what a lazy scheduler will do.

The alternative to a lazy scheduler is to allow background tasks, ie, processes that execute when no real-time processes are ready. However, it can be argued that all processes have some kind of deadline constraint associated with them; if it was possible to postpone a process indefinitely, then it would not need to be included in the system. In this sense, omitting a deadline specification for a process does not mean the process does not have a deadline, but rather that the programmer does not know, or does not care, what it is. For better or worse, forcing the programmer to assign explicit deadlines, also to processes that would otherwise be implemented as background tasks, means that their actual timing requirements can no longer be ignored.

A lazy scheduler also provides a form of fail-fast approach to the specification of deadlines. Without a lazy scheduler, missing a deadline specification for a task, or incorrectly considering a task to be of background priority, is an error that in most situations will not be detected; the effects may only become apparent when the system load is high. With a lazy scheduler, a task with no deadline will never be executed, and any testing of that task will immediately fail.

An argument for allowing background tasks is that some processes are so much more important than others that the latter should be dropped entirely when the load on the system is high. Still, completely disabling a process during high loads may have serious consequences, even if the affected processes are considered non-critical, and suitable for implementation as background tasks. For example, a process that performs system logging has been described as non-critical [29], but, if one assumes that the system meets its deadlines under normal conditions, then the extraordinary situation where not all deadlines will be met may be exactly the kind of situation where system logging is needed the most.

4.4.2 Implementation of Laziness

In practice, there must be some exceptions to the laziness rule; for example, control and loop structures that contain **TIME** constructs should not require separate deadlines of their own, or it would become difficult to properly implement conditional or periodic tasks. The precise definition of the laziness of Toc is instead the following:

Definition 4.1 (Toc Laziness)
In Toc, no *primitive processes* except **SKIP** are executed unless needed to complete a process with a deadline.

The restriction to primitive processes means that compound process constructors are exempted from the laziness rule, and are allowed to execute until reaching an inner primitive process. This restriction is necessary to allow the **TIME** constructs themselves to be evaluated: A periodic process can then be created by wrapping a **TIME** construct in a **WHILE**, without needing to set a deadline for the **WHILE**.

The laziness rule prevents non-primitive processes from being executed without a deadline, but it does not imply that code executed without a deadline only requires insignificant execution time. For example, an arbitrarily complex expression may be used as the condition in a **WHILE** construct. It does mean, however, that no code with side effects will be executed without a deadline, and consequently, that all functionality that manipulate input or output from a program will need to be given a deadline.

The exception for **SKIP** is necessary to satisfy the CSP law that **SKIP**

Listing 4.4: *Conditional Task that Requires Non-Lazy* ***SKIP***

```
SEQ
  IF
    p
      TIME t.1
        P()
    TRUE
      SKIP
  TIME t.2
    Q()
```

is an identity process for the sequence operator [88], ie, that the semantic meaning of a process does not change if a **SKIP** is inserted in sequence before or after it. Moreover, a practical benefit of making an exception with **SKIP** is that it simplifies the writing of conditional **TIME** constructs:

In occam, conditionals may have many branches, and be nested and replicated, but the behavior of a conditional process must be defined for all cases: In contrast to most other imperative languages, where conditionals typically default to a behavior similar to **SKIP** when no predicates evaluate to true, occam **IF**s defaults to **STOP**, which deadlocks the program. A default behavior of **SKIP** is instead achieved by appending an explicit **TRUE**–then–**SKIP** branch to the conditional.

As a consequence, if **SKIP** was lazy, ie, required a deadline to execute, a conditional **TIME** construct that was not selected would prevent the evaluation of **TIME** constructs in sequence to the conditional. For example, the conditional task in listing 4.4 would then not work properly: unless the whole process had a deadline, the second **TIME** construct would only be executed when *p* was true. By exempting **SKIP** from the laziness rule, the second **TIME** construct will always be executed.

For symmetry, it would seem appropriate to exempt **STOP** from the laziness rule as well. **STOP** is mostly used in error situations, either explicitly as one may use the abort() function in C, or implicitly, such as the equivalent process to a deadlock, or to an **ALT** where all guards are closed by **FALSE** Boolean expressions. A lazy **STOP** will only halt the program if it has a deadline for its completion, while a strict (ie, non-lazy) **STOP** will halt the program when it becomes eligible for execution. This topic has not been given much consideration, and in the current version of the language, **SKIP** is the only strict primitive process.

4.4.3 Discovery and Base Times

The Toc run-time schedules strict code using a mechanism for partial execution called discovery. Discovery is started on processes without a deadline when there is a chance to reach a **TIME** construct without having to execute any primitive processes except **SKIP**. Discovery stops when a primitive process except **SKIP** is encountered.

Discovery is necessary to allow the Toc scheduler to be aware of all the deadlines in the system, and needs only to be started whenever there is a chance to uncover a **TIME** construct that may begin execution. This limits the need for discovery to the following situations:

1. At the start of a program, on procedure Main.

2. After a **TIME** construct is completed, on processes in sequence.

3. After a **HANDLE** construct accepts an event, or after a **TIMEOUT**.

4. After a channel communication, on processes in sequence at both ends.

If a **PAR** is discovered during discovery then the discovery will continue on all branches of that **PAR**.

The base time of **TIME** constructs found during discovery is set to the time of the event that caused the discovery. For the situations listed above, the base time is set, respectively, to

1. The startup-time of the program.

2. The deadline of a newly completed **TIME** construct if the deadline was met, or its actual completion time if the deadline was missed.

3. The time when an event was raised, or the time when the timeout expired.

4. The time when the communication rendezvous was completed.

These rules satisfy the required properties of time-constructs set out in section 3.4 (properties 3.1 to 3.4).

Information from the compiler can be used to skip discovery in cases where it can be known in advance that no **TIME** constructs will be discovered, for example if the next process in sequence is a primitive process that is not **SKIP**.

4.4.4 Deadline Inheritance over Channels

occam 2.1 does not associate notions of urgency with its communication. If a process wishes to communicate on a channel that is not ready, then the process will be blocked and another non-deterministically chosen process will execute instead. In a real-time system this kind of behavior would quickly lead to priority inversions, and some means for associating urgency with communication is therefore necessary. In Toc this is achieved by deadline inheritance through channels.

As noted in observation 3.5, all synchronization protocols require that whenever a process is blocked, it must be possible to identify which process is blocking it. In Toc this is ensured by the usage rules: A channel that is used for input or output by a process cannot be used for the same by any other processes in parallel. The Toc compiler enforces these rules at compile-time, and while checking that each channel is only used for input and for output by one process at a time, it also creates an expression for *which* process use that channel. This expression is then used run-time to find out, when a process wants to communicate on a channel, which process holds the other end and should inherit its deadline.

The process whose execution leads to the next communication on an end of a channel is said to own that end of the channel:

Definition 4.2 (Channel Ownership)
The input (output) owner of a channel is the process whose execution will lead to the next input (output) on that channel.

The initial input and output owner of a channel is the process following the declaration of the channel. Channel ownership is updated run-time at the beginning and end of every **PAR**, using information gathered by the compiler during the usage rules check.

Deadline inheritance is implemented in the scheduler by allowing processes to "forward" execution: If a process with a deadline requires communication with another process, then every time the first process is allowed to execute, it will instead perform a context switch to the second process. This has the same effect as deadline inheritance, with the second process inheriting the deadline of the first. When channel ownership is known, the desired behavior of the forwarding mechanism may be defined as follows:

Definition 4.3 (Deadline Inheritance over Channels)
If the current process needs to complete an input (output) on a channel that is not ready, then forward execution to the output (input) owner of that channel.

Per definition, the owner is the process whose execution will lead up to the next communication on that channel, so this forwarding is the fastest way of completing the communication. The forwarded process is executed up to the rendezvous point; then the forward is canceled and execution continues from the side with the earliest deadline.

4.4.5 Timing-Aware Alternation

In occam 2.1, if multiple guards are open in an alternation, then one may be chosen non-deterministically, although as noted earlier, most implementations of alternation are completely unfair and will always select the textually earlier process. In Toc, the choice should be based on deadlines whenever possible. Three separate cases can be identified:

1. At least one communication guard is open, where the owner of the other end has an associated deadline.
2. At least one guard is open, but there are no communication guards where the owner of the other end has a deadline.
3. No guards are open.

In the first case, the choice of alternative is simple: it is the guard where the process owning the other end of the channel has the earliest deadline. This situation occurs eg, when the **ALT** has inherited the deadline from the owner of the channel being selected. Note that if the program satisfies observation 3.6, then this first case is the only one that will occur.

In the other cases the choice is less obvious. The **ALT** itself has a deadline to meet, or it would not be executed, but in complex programs it is not always feasible to determine which alternative that best helps it meet that deadline. Therefore, if at least one guard is open, but no guards have owners with associated deadlines, a guard may be selected non-deterministically. The current implementation always selects the textually earlier open guard in this case.

However, there may not be any open guards at all. In occam, the alternation process will wait if no guards are open, but in Toc the other processes may be lazy, so merely waiting is not an option. Instead, if there are no open guards then the **ALT** must forward execution to some process which owns an end of a channel used in one of the guards, until that, or another guard, becomes open.

In both Toc and occam, if all guards have a closed Boolean condition then the **ALT** will act as a **STOP**.

Listing 4.5: *Lazy Forward*

```
  WHILE TRUE
    INT x:
    SEQ
      input ? x
      output ! x
:
```

Listing 4.6: *Strict Forward*

```
  WHILE TRUE
    INT x:
    input ? x
      output ! x
:
```

occam 2.1 restricts communication guards to inputs because of the implementation difficulty associated with resolving mutually communicating alternations. An efficient algorithm for implementing this has now been developed [76], so this restriction could possibly be relaxed. In Toc, output guards are often practical and are therefore permitted. However, mutually communicating **ALT**s are still not supported and may silently deadlock; this is a limitation of the current implementation.

4.4.6 Use of Extended Rendezvous

Laziness complicates the writing of multiplexers and simple data forwarding processes. An illustration is given in listing 4.5: If something arrives on channel 'input' then it will not immediately be passed on to channel 'output' because there is no deadline driving the execution of the second communication. The output will first happen when driven by the deadline of the next input, as the latter cannot proceed before the former has completed. An extra **TIME** construct can be added to perform the output with a new deadline, but this could cause a stall for the next process to use the 'input' channel. Moreover, the deadline of such a **TIME** construct will either be arbitrary or a repeated specification of an existing deadline, neither of which are desirable from a quality point of view. Another undesirable solution would be to avoid forwarding data altogether, but this would put severe restrictions on program organization.

The extended rendezvous feature of occam-π [104] solves the problem. In Toc, when using an extended rendezvous then the deadline driving the communication must also drive the during-process, because its completion is required to complete the communication. An example of the correct way to forward data is shown in listing 4.6.

occam 2.1 only supports extended inputs, which use a double question-mark (`??`) operator. Toc supports both extended inputs and outputs, but does not use any special operators for it. The existence of the indented during-process is instead used to identify the communication as extended.

This is unproblematic for single communications, but has a notable effect on alternations as there is already a process indented under the communication. A consequence of this is arguably the most important syntactic difference between occam and Toc:

> In Toc, communications in alternations are extended rendezvouses *by default.*

If a non-extended rendezvous is desired in an alternation, the during process must be set to **SKIP** and the alternative process written below it; eg,

```
ALT
  request ? x
    SKIP
    reply ! y
```

There are drawbacks both of making extended rendezvous the default, and not making it the default, each which reduce the readability of certain programs.

The drawback of keeping occam notation, and not making extended rendezvouses the default for alternations, is to introduce a type of timing-related programming error that is hard to spot. For example, say that a process reads from channels in an **ALT**, and is supposed to do some work on incoming data under the deadline of the sender, as part of an extended rendezvous. However, the extra question mark is missing, so that the rendezvous is instead non-extended. The consequence is that each request will be handled under the deadline of the next request. If requests are frequent then this is an almost silent error that is hard to trace in a large system, and is made worse by being caused by an almost invisible typo.

The drawback of instead making extended rendezvous the default, is another type of almost invisible error, one that will cause the system to deadlock. This occurs when the alternating process performs an explicit reply through another channel. The reply can not be part of an extended rendezvous, as this will cause a circular wait and deadlock. Therefore, a **SKIP** must be inserted as the during process, with the reply being the non-extended part of the action (like in the above example). This is a **SKIP** that may easily be forgotten.

The reason for choosing the second kind of error over the first is that a system with the second type of error will never work; it will deadlock at every request of that type made to the server. The first kind of error, on the other hand, is subtle and may hardly be noticeable if there are frequent requests, as the output of one request will be handled with the deadline of the next.

4.5 The Toc Compiler and Run-time System

A brief overview of the implementation of the compiler and run-time system will be given here, together with a list of limitations of the current version.

Note: It is worth pointing out that neither the compiler nor the run-time system was written with performance in mind; performance has been sacrificed for simplicity of implementation whenever possible. The compiler and run-time system were written in order to experiment with new language primitives, rather than to improve existing primitives, and performance was therefore not considered an important feature. In particular, the run-time system adds its own preemptive scheduler on top of an existing OS scheduler, which significantly increases task switch times compared to a more efficient run-time.

4.5.1 Compiler Overview

The Toc compiler (`tocc`) is a six-stage compiler, with the five first stages written in Haskell, and the final stage performed by an external C-compiler. The Haskell sources utilize Glasgow-extensions [77], and have only been tested with the Glasgow Haskell Compiler (`ghc`). The output from the fifth compiler stage is standard ANSI C99, but has only been tested with `gcc`.

The Toc language definition is written in a format called LBNF, which is processed using the multi-language parser generator `bnfc` [81]. This produces lexer and parser definitions, which in turn are translated to Haskell code by `alex` and `happy`—the Haskell equivalents of `lex` and `yacc`—respectively. Layout is resolved between lexing and parsing, by annotating the token stream with '`{`', '`}`' and '`$`'-symbols, denoting the beginning of a block, the end of a block, and the end of a line, respectively. The layout rules are described in detail in appendix A. The main compiler stage takes a Toc syntax tree as input and generates a C syntax tree as output, which is then written to a file using a C pretty-printer. The C-file is compiled by `gcc` and linked with the run-time library to create an executable. As `gcc` is supported by a large variety of platforms, this allows the Toc compiler to produce reasonable platform-independent programs. An overview of the process is given in fig. 4.1.

The design of the main compiler stage is fairly straight-forward and will not be discussed in detail. Each parallel branch is converted to a function so that it can be executed separately; the compiler also keeps track of a tree of variable scopes so that different instances of the same branch—these are created when replicating parallels—get a unique copy of their own variables,

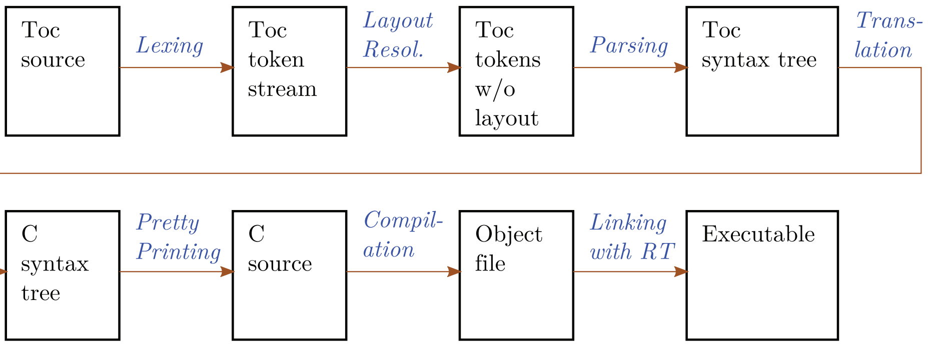


Figure 4.1: *Overview of Compiler Stages in* `tocc`

but share variables from their common stem. Whenever a variable is used, it is flagged by the type of use (eg, read, output, abbreviate). Usage rules are evaluated when a variable goes out of scope.

The usage rules are fairly simple to evaluate for scalars. With arrays, the indexing expressions may also have to be analyzed if correct use cannot be ensured without taking the indexes into account. In that case indexes will be limited to expressions of constants, literals and replicators (variables defined by a **FOR**). The compiler will, if necessary, simulate all required replicators to determine which indexes of an array that are used by any given process. Better, but more complex methods exist, such as the Omega Test [83], which has been used by other occam compilers to do usage rules checks analytically, rather than by simulation [23].

4.5.2 The Run-Time System

The run-time system is written in C/POSIX, and uses POSIX-threads to manage parallel processes. This limits the number of simultaneous parallel processes that can exist compared to a system using light-weight processes, and increases the overhead associated with switching between them. Ideally, the run-time system should be replaced with a system that supports light-weight processes; however, these are generally not preemptible, which is one reason why they achieve low overhead and fast switching times. If one were to use light-weight processes in a real-time system one would have to find a balance between the low overhead and fast switching times gained from non-preemptible light-weight processes, with the responsiveness that is achieved by having processes fully preemptible.

The run-time system performs its own scheduling of processes to accommodate for laziness and EDF. It is organized as three modules, which are

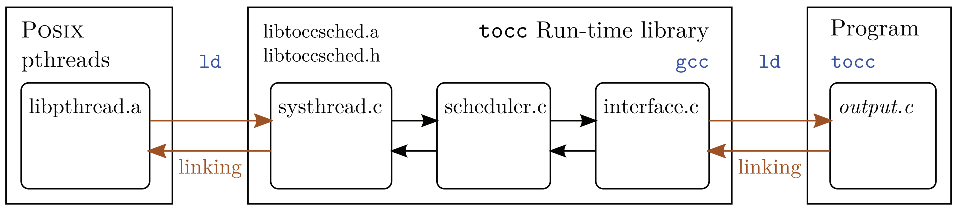


Figure 4.2: *Overview of the `tocc` Run-Time Library*

structured as shown in fig. 4.2.

The first module, `systhread`, manages operating system threads in such a way that at most one thread executes at any given time. It thus allows the system to execute its own scheduler under an existing OS. POSIX signals are used to suspend and resume threads, but threads may also voluntarily suspend themselves. This is the only module that depends on POSIX threads; by replacing this module it should be possible to port the existing run-time to non-POSIX compliant operating systems as well.

The second module is `scheduler`, which implements the lazy EDF scheduler. In addition, it provides functionality for forwarding execution, so that the currently executing process can ask for another process to be executed in its place. This module also manages discovery, which is handled by temporarily giving processes zero relative deadline. Discovery ends when the processes call interface_end_discovery(), these calls are inserted by the compiler before primitive processes that are not **SKIP**.

The third module is the `interface`, which contains functions that are called by the compiled Toc programs. The interface contains run-time support for primitives such as alternations, the creating of parallel processes and communication over channels, and translates this functionality into requests to the scheduler.

4.5.3 Examples of Translated C-code

A few examples will be presented to show the structure of the translated C-code; these are likely to be more instructive than presenting further details of the compiler itself. The examples show the output from the fifth compiler stage (the input to `gcc`) when compiling the Toc program in listing 4.7. The output has been filtered through an automatic code formatter (`indent`) to improve readability, and a pair of superfluous curly braces was removed to save space.

The translated main() function is shown in listing 4.8. It zero-initializes

Listing 4.7: *Translated C-Code: The Toc Source*

```
PROC Main()
  CHAN INT channel.1, channel.2:
  PAR
    TIME 1 SEC
      SEQ
        PRINT "a"
        channel.1 ! 42
    TIME 2 SEC
      SEQ
        PRINT "c"
        channel.2 ! 42
    INT rv:
    WHILE TRUE
      ALT
        channel.1 ? rv
          PRINT "b"
          PRINT "d"
        channel.2 ? rv
          PRINT "e"
:
```

a scope, and sets itself as the input and output owner of all new channels. Then, a parallel structure is created and its branches are initialized with function pointers to their start-up processes and the parent scope. The interface_activate_par() function will forward execution to a child branch until all of them have terminated.

The first of the branches with a **TIME** construct is shown in listing 4.9. After signaling to the run-time system that a new parallel process ("thread") has been created, it creates its own scope, and takes the sending side of the channel that it will use later. The **TIME** construct is then initialized. After that, a primitive process is encountered, and a call to interface_end_discovery() is therefore made; this call does nothing if the process already has a deadline.

The process then outputs 42 on channel.1. This requires two calls to the run-time library. First, interface_pre_write_channel() is called to indicate that the sender is ready; this call will block until the receiver have rendezvoused, then it returns with the memory address where the receiver wishes to store the data from the channel. The run-time system will forward execution to the receiver—identified by being the input owner of the channel—if it is not yet ready to communicate. After writing the data, the sender calls interface_post_write_channel(). This is the last action to be part

Listing 4.8: *Translated* C*-Code: The main() function*

```
void Main()
{
    struct __scope_1_t __scope_1 = { {0}, {0} };
    interface_channel_take_receiver(&__scope_1.channel_1);
    interface_channel_take_sender(&__scope_1.channel_1);
    interface_channel_take_receiver(&__scope_1.channel_2);
    interface_channel_take_sender(&__scope_1.channel_2);
    {
        struct par_t par = { 0 };
        struct thread_t threads[3] = { {{0}} };
        interface_init_par(&par, threads, 3);
        interface_set_par(&par, 0, (void *(*)(void *))__par_2_0, &__scope_1);
        interface_set_par(&par, 1, (void *(*)(void *))__par_2_1, &__scope_1);
        interface_set_par(&par, 2, (void *(*)(void *))__par_2_2, &__scope_1);
        interface_activate_par(&par);
    }
}
```

Listing 4.9: *Translated* C*-Code: A **TIME** Construct*

```
void __par_2_0(struct __scope_1_t *__scope_1)
{
    interface_start_thread();
    struct __scope_3_t __scope_3 = { __scope_1 };
    interface_channel_take_sender(&__scope_3.parent->channel_1);
    {
        struct timespec newtime = { 0 };
        struct timeblock_t timeblock = { 0 };
        newtime = timespec_sec(1);
        interface_init_timeblock(&timeblock, newtime, 1);
        interface_end_discovery();
        printf("%s", "a");
        printf("\n");
        interface_end_discovery();
        {
            int *__tocc4 =
                (int *)interface_pre_write_channel(&__scope_3.parent->channel_1);
            *__tocc4 = 42;
            interface_post_write_channel(&__scope_3.parent->channel_1, false);
        }
        interface_end_timeblock(&timeblock);
    }
    interface_channel_give_sender(&__scope_3.parent->channel_1);
    interface_end_thread();
}
```

of the **TIME** construct, which it is then deleted, canceling the deadline of the process. Finally, the process passes ownership of its channel end to its parent and terminates. The branch with the second **TIME** construct is near identical to the first, and will not be shown.

The alternating process is shown in listing 4.10. It starts like the other two processes, by initializing a scope and taking ownership of its channel ends. The alternation is implemented as a **while**-loop, cycling through its guards and calling interface_try_alt() on each, with the channel and the Boolean expression as arguments (this is the inner loop, the outer is the **WHILE** from the Toc source). When this function returns true, then that alternative should be selected; a **goto** at the end of each action will then exit the loop. Otherwise, if no alternative was found appropriate, then interface_retry_alt() is called. This function makes the **ALT** less particular for the next iteration of the loop; the conditions for accepting an alternative follow the progression described in section 4.4.5. If all alternatives are blocked by a Boolean guard, then the **ALT** is equivalent to a **STOP**, and will terminate the program. This will show up as a run-time error describing the error and the source, which is file `text.tocc`, line 15, column 9.

Each alternative reads from a channel. Again, two run-time system calls are required: interface_read_channel() and interface_post_read_channel(). To the first function, the address of an immediate variable is passed, which the sender uses to write its data. The during-process is then executed, before the second system call.

The examples of translated C-code given above illustrate the most Toc-specific parts of the C-code generation in the compiler. The translation of other language features, such as those relating to basic computation or control structures, are more straightforward, and further examples will not be given.

4.5.4 Limitations in the Current Version

The current implementation is a prototype, and has several limitations. First, due to limitations in the parser generator, there are minor syntactic differences between Toc and occam. These are to be considered shortcomings of the current implementation, rather than changes made for their own sake.

Some differences are minor; eg, while occam uses asterisks ($*$) as the escape character in strings, Toc uses backslashes. However, one difference in particular is significant: Arrays in occam are declared with size brackets to the left to improve readability; the declaration

Listing 4.10: *Translated* C-*Code: The **ALT***

```
void __par_2_2(struct __scope_1_t *__scope_1)
{
    interface_start_thread();
    struct __scope_7_t __scope_7 = { __scope_1 };
    interface_channel_take_receiver(&__scope_7.parent->channel_1);
    interface_channel_take_receiver(&__scope_7.parent->channel_2);
    struct __scope_8_t __scope_8 = { &__scope_7, 0 };
    while (1) {
        struct alt_t alt = { 0 };
        interface_init_alt(&alt);
        interface_end_discovery();
        while (1) {
            if (interface_try_alt(&alt, 1, &__scope_8.parent->parent->channel_1)) {
                int __tocc10;
                interface_read_channel(&__scope_8.parent->parent->channel_1,
                                       &__tocc10, true);
                __scope_8.rv = __tocc10;
                interface_end_discovery();
                printf("%s", "b");
                printf("\n");
                interface_post_read_channel(&__scope_8.parent->parent->channel_1);
                interface_end_discovery();
                printf("%s", "d");
                printf("\n");
                goto __alt_9_alt_finished;
            }
            if (interface_try_alt(&alt, 1, &__scope_8.parent->parent->channel_2)) {
                int __tocc11;
                interface_read_channel(&__scope_8.parent->parent->channel_2,
                                       &__tocc11, true);
                __scope_8.rv = __tocc11;
                interface_end_discovery();
                printf("%s", "e");
                printf("\n");
                interface_post_read_channel(&__scope_8.parent->parent->channel_2);
                goto __alt_9_alt_finished;
            }
            interface_retry_alt(&alt, "test.tocc", 15, 9);
        }
    __alt_9_alt_finished:
        interface_end_alt(&alt);
    }
    interface_channel_give_receiver(&__scope_7.parent->channel_1);
    interface_channel_give_receiver(&__scope_7.parent->channel_2);
    interface_end_thread();
}
```

[4]**CHAN** [2]**INT** array:

declares four channels each using a two-element integer array as data type. Also, in occam it is possible to create an abbreviation without a type, eg,

a **IS** b:

In Toc, however, due to limitations on lookahead in the parser generator, all declarations are required to start with a keyword or an identifier that could not have been part of an expression. Therefore, abbreviations require the type, even though it is implied by the right-hand side; and array declarations must have the brackets on the right side. The above array declaration thus becomes

CHAN[4] **INT**[2] array:

For similar reasons, procedure names in Toc must use identifiers that start on an upper-case letter, and is immediately followed by a non-uppercase letter.

Another limitation is that Toc does not have any kind of IO interface built into the language. Input and output must instead be handled using external functions; these can be declared with the **EXTERN** keyword and must be linked into the program using command line arguments to the compiler.

There is also some functionality in the compiler that has not been implemented and a few known bugs, including

- A **VALOF** cannot be used, except as the first process in a function.
- The current time **NOW** can be used in functions, even though it is not pure.
- The usage checker is overly tolerant in some cases involving abbreviations and procedures.
- Expressions are translated directly to C; therefore, all arithmetic operators will overflow, like C, not raise errors as in occam.
- No dynamic count is allowed in a **PAR FOR**; a constant expression is required.
- Programs with mutually communicating alternations will compile, but may deadlock at run-time.

4.6 Case Study: Elevator Control

One of the systems that have been implemented in Toc is a control system for a miniature elevator model. This implementation will be used to demonstrate programming in Toc, and key parts of the implementation will be presented and discussed.

4.6.1 Hardware Setup

The system hardware consists of a four-floor elevator model, with a motorized sliding metal plate symbolizing the car of the elevator. Proximity sensors provide feedback for when the car is at a floor, and a button panel provides input from the user. The buttons also have lights on them to provide feedback to the user. The model does not have a mechanical door; instead, a light is used to indicate that the "door" is open.

In total, the inputs from the elevator consist of 11 buttons, one toggle switch and four proximity sensors:

1. Six call buttons that represent buttons external to the elevator: three "call-up" buttons for floors 1 to 3 and three "call-down" buttons for floors 2 to 4.
2. Four "go-to" buttons, representing buttons inside the elevator.
3. A "panic" button.
4. One "door obstruction" toggle switch, which if turned on represents something obstructing the door.
5. Four proximity sensors, one for each floor, to detect the position of the elevator when moving.

The outputs are:

1. 11 button lights, one on each button.
2. A light representing whether the door is open or not.
3. An analog output to a motor. However, the motor is sufficiently geared down to be used at maximum speed without the need for further control.

The elevator is connected to a computer through a DAC interface card that is controlled using the `comedi` library [90]. All inputs must be polled as there are no interrupts or similar mechanisms that notify the system of a change in input state. The host computer runs a standard Ubuntu Linux distribution.

4.6.2 Software Specification

The elevator should behave like an ordinary elevator and serve requests according to the standard elevator algorithm; that is, it should only change direction at the end of its range, or if nobody is waiting or wants to go further in the current direction.

Lights on buttons should represent pending requests of that type. If a user is waiting to go in one direction, and the elevator stops at that floor, heading in the opposite direction, then the user's request should not be served.

Pushing the panic button should automatically clear all queues and stop the elevator, and also prevent the door from closing. This panic state may only be canceled by pushing one of the goto buttons inside the elevator.

Opening and closing the "door" is assumed to take 1 s. If users are entering or exiting at a floor, then the elevator should stop and keep the doors open for at least 10 s, before closing the door and performing the next appropriate action. The closing of the door may be delayed by the obstruction switch, or by an additional request for the same floor: the doors should close 10 s after the last time the obstruction switch was turned off or there was a request to the current floor.

It was also discovered that in order to stop at a floor—and not drift past it—the motor direction has to be reversed a short time as a means of breaking. A suitable duration was found to be 5 ms.

Design of Temporal Specifications

The inputs to the system are polled, and intuitively, polling is best handled using an implicit deadline periodic task. The polling period is a temporal constraint that depends on the minimum duration of the inputs to be polled, eg, the minimum duration of a button press, but also on the response time requirements from sensor stimuli to system response. For this system, a polling period of 20 ms was thought to be appropriate, which guarantees a maximum of 40 ms between two consecutive polls (observation 3.2).

In a traditional real-time system, the 20 ms implicit deadline of the polling task would be the only deadline in the system, and all the responses to polling stimuli would be driven by this deadline. The other temporal constraints—the 5 ms breaking duration, 1 s for opening and closing doors and the 10 s for allowing users to walk on or off—would be implemented using delays and timeouts.

One issue with this approach is that the only deadline constraint in the system is derived from an implementation detail—polling—and not from the behavioral requirements of the system as seen from its users. It is possible to implement the system using only the polling deadline also when programming in Toc, but it is more natural to use the polling deadline to drive only the polling, and instead trigger other deadlines when stimuli are detected.

In this way, one becomes free to specify temporal constraints for the system as one sees fit. As Toc is a lazy scheduled language, every action performed by the elevator must be given a deadline, or be driven by another action with a deadline; the programmer must therefore explicitly consider which temporal constraints apply to each function the software is required to perform.

Ideally, the set of necessary temporal constraints would be uniquely given by the problem to be solved, but in practice there is a great deal of flexibility in how to specify these constraints. For example, in the following implementation it was chosen not to specify a separate deadline for turning on button lights in response to a button push; this is instead driven indirectly by the polling deadline.

The set of temporal constraints also depends on the organization of modules in the system, as there may be a conflict between the desire to specify system-wide deadlines, and the desire for low coupling between modules. For instance, say that the system should complete system-wide initialization within a certain time. However, the code for initializing a module typically resides in that module, so in order to implement a system-wide initialization deadline one must either centrally organize initialization, tightening coupling, or repeat the initialization deadline in each module, weakening cohesion.

It is also necessary to consider how to avoid stalls and deadlocks when designing the set of temporal constraints. If a process communicates with another process using a channel, then deadlines in the former may drive actions in the latter, but the latter should not contain delay constraints or the system may stall (observation 3.7). Stalls may be avoided by replacing channel communication with events, but then the event handling process must specify its own deadline, so this change will affect the temporal specifications of the system. Similarly, the raising of an event will never block, so events may replace channel communications as a method for deadlock avoidance. However, this will also require a new deadline for handling the

Table 4.3: *Elevator Example: List of Temporal Constraints*

No.	Description	Type	Value
1	Polling interval	Impl. deadl. peri. task	20 ms
2	Reversing the motor	Delay	5 ms
3	Initialization	Deadline	10 ms
4	Panic event	Event handler	10 ms
5	End-of-panic event	Event handler	1 s
6	Wake up from idle	Event handler	1 s
7	Floor sensor event	Event handler	100 ms
8	Opening the door	Deadline	1 s
9	Waiting to close the door	Timeout	10 s
10	Obstruction event	Event handler	10 ms
11	Closing the door	Deadline	1 s

event, affecting the set of temporal constraints.

4.6.3 Implementation Overview

For the elevator implementation it was chosen to divide the system into the following main modules, referred to by their procedure names:

1. **PROC** FilteredIO, responsible for IO.

2. **PROC** Mover, handling the motor and sensors.

3. **PROC** Queue, handling button presses and queuing logic.

4. **PROC** Door, handling obstructions, and opening and closing of the door.

The organization is shown in fig. 4.3. In the following text, the most notable parts of the implementation will be discussed, including the implementation of all the temporal specifications, and how the choice of module composition affected the design of these specifications.

The full list of temporal constraints is given in table 4.3. Each of the constraints corresponds to a **TIME** or **TIMEOUT** construct in the implementation source code.

Note: In the following listings, some code is removed and replaced with "..." due to space considerations. The full source code for the elevator implementation can be found in appendix B.

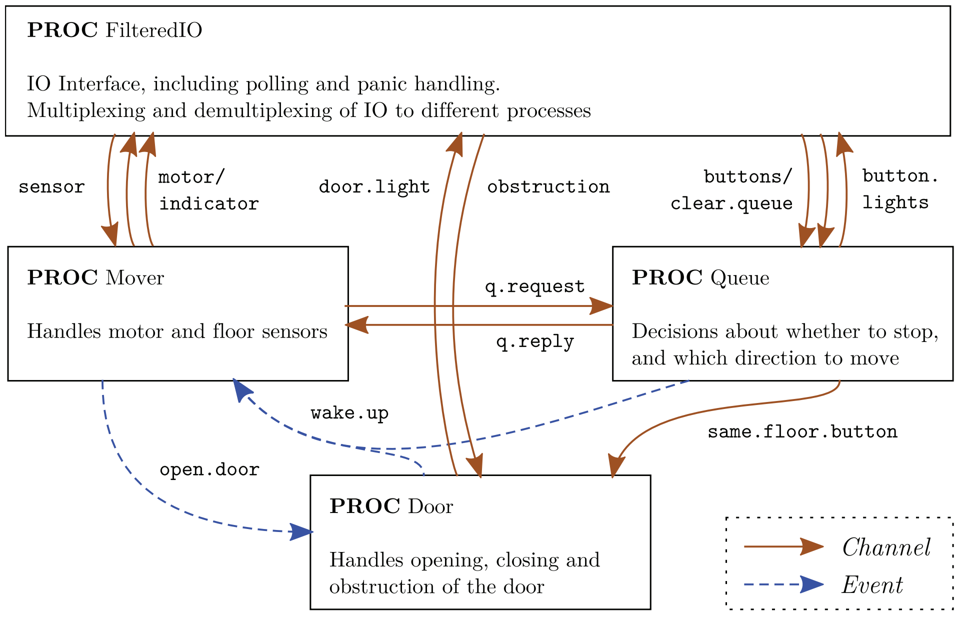

PROC FilteredIO
IO Interface, including polling and panic handling.
Multiplexing and demultiplexing of IO to different processes
sensor
motor/ indicator
door.light
obstruction
buttons/ clear.queue
button. lights
PROC Mover
Handles motor and floor sensors
q.request
q.reply
PROC Queue
Decisions about whether to stop, and which direction to move
wake.up
same.floor.button
open.door
PROC Door
Handles opening, closing and obstruction of the door
Channel
Event

Figure 4.3: *Elevator Example: Overview of Processes*

4.6.4 The FilteredIO Process

An observation made during the early stages of the implementation was that the code for handling panics was almost orthogonal to the code that handled non-panic situations, and that the latter code could be greatly simplified by moving the handling of panics into a separate process.

The FilteredIO process was designed to implement this; it handles panics, as well as system IO, in such a way that the functionality of the rest of the system can be implemented without taking panic into account. For example, in panic state, button presses will be discarded instead of being passed down to the application, and motor commands passed up from the application will be ignored. A visual overview of FilteredIO is given in fig. 4.4. For simplicity, the names of internal channels are not shown.

The IO.Input Process

The IO.Input process contains the polling functionality, and the polling period temporal constraint. A fragment of it is shown in listing 4.11, where the **WHILE**...**TIME** arrangement can be identified as implementing an implicit deadline periodic task. The listing also shows the detection of a button push for one of the goto buttons inside the elevator. When detecting a change

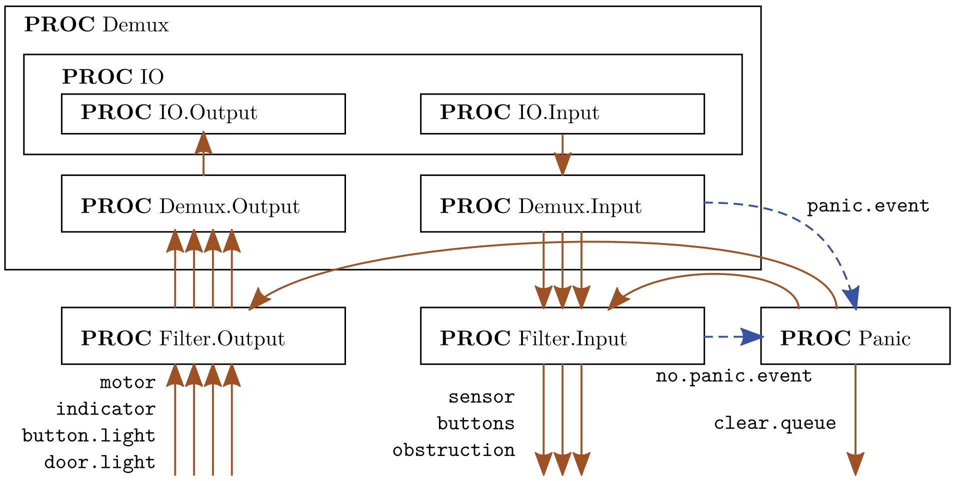


Figure 4.4: *Elevator Example: Overview of the FilteredIO Process*

Listing 4.11: *Elevator Example: The IO.Input Process*

```
PROC Io.Input(VAL INT handle, CHAN IO.INPUT input)
  BOOL [floors] call.down.state:
  ...
  WHILE TRUE
    TIME 20 MSEC -- Polling interval
      SEQ
        ...
        SEQ i = 0 FOR floors
          SEQ
            Io.Read.Bit(handle, port.goto[i], bit)
            IF
              bit AND (NOT goto.state[i])
                input ! goto ; i
              TRUE
                SKIP
            goto.state[i] := bit
        ...
:
```

in state from not-pushed to pushed, a message is sent through the “input” channel.

The use of channels rather than events to communicate inputs means that the receiving process may choose whether or not to add its own temporal constraint for handling the input. For example,

- by reading from the input channel using an extended rendezvous, the receiving process can handle the input using the deadline propagated through the channel, ie, the polling deadline.

- By using a non-extended input it can simply store the information of the input, delaying its handling until required by another deadline.

- By raising an event as a response to the input it can trigger a sporadic task, which then assigns its own deadline to handling the input.

If the IO.Input process had raised an event to signal new input rather than communicating it through a channel, then the receiving process would have had no choice but to handle the input using a sporadic task. An advantage of this, on the other hand, would have been looser temporal coupling between IO.Input and the receiving process; as it is now, the polling process may miss deadlines if the receiving process is too slow in handling the input.

The IO.Output Process

Output from the system is handled by the IO.Output process. It too communicates with external processes using channels, and sets its outputs using extended rendezvouses. This way, it can function as a passive library with respect to real-time constraints, and let the various processes that require output decide the urgency. The alternative, of using events to communicate outputs, would mean that the IO.Output process would have had to set its own deadlines for applying output.

A section of the IO.Output process is shown in listing 4.12. The process reads each output command from a single channel, and processes the commands using the deadline from the implicit extended rendezvous.

When a motor speed of 0 is requested, the motor direction is reversed for 5 ms before the motor is stopped. Because of the extended rendezvous, the channel communication is not considered complete before this process has finished, and the sender of the motor command will therefore also be delayed 5 ms.

The Panic Process

IO.Output and IO.Input are connected to a multiplexer and demultiplexer process, respectively. As channels are one-to-one only, this is necessary to allow multiple processes access to IO. The input demultiplexer has a special handling of panic button presses, which raises a panic event rather than being passed on (fig. 4.4).

When the system is in panic state, the elevator must be stopped, and all further commands be ignored, until a user presses a goto button inside the elevator. The queue should also be cleared. By raising a panic event, rather than sending a panic button push through a channel, one must assign an explicit deadline to handling the panic button press, distinguishing it from other button presses.

End-of-panic is also raised as an event, and thus also requires its own temporal constraint. In this case it might have been more intuitive not to specify a constraint, but use the polling deadline instead. However, this would cause a circular graph of communications (see fig. 4.4) which would make it harder to prove that the system is free from deadlocks.

The implementation of the Panic process is made somewhat complicated due to the lack of an alternation process that accepts events. This must instead be accomplished using two **HANDLE** constructs in parallel. As parallel branches are not allowed to share variables or channel ends, operations that are common to the handling of both a panic and an end-of-panic event must be placed in a separate parallel process. The implementation (listing 4.13) therefore contains three parallel branches; two event handlers, and a common server used by the handlers.

4.6.5 The Mover Process

The Mover process (fig. 4.5) is responsible for starting and stopping the motor. It is implemented as two mutually exclusive processes, Mover.Idle, which is executing only when the elevator is standing still, and Mover.Moving, which is executing only when the elevator is moving.

The Idle and Moving processes are quite similar (the latter is shown in listing 4.14). Each handles an event; the former handles the wake.up event and the latter handles the sensor event. In response to the event, they both request the current floor from a common server, and send a what.now request to the Queue process, stating which floor they are at, and which direction they were going before they got there. The Queue process responds with either “move”, “sleep” or “open”, based on the state of the queue. Note that because Toc accepts outputs in alternations, the data direction between

Listing 4.12: *Elevator Example: The IO.Output Process*

```
PROC Io.Output(VAL INT handle, CHAN IO.OUTPUT output)
  ...
  WHILE TRUE
    ...
    output ? CASE
      goto.light ; floor ; setting
        Io.Set.Bit(handle, port.light.goto[floor], setting)
      ...
      motor.speed ; speed
        IF
          speed = 0
            SEQ
              Io.Set.Bit(handle, port.motordir, NOT previous.motor.dir)
              TIME 5 MSEC -- breaking duration
                SKIP
              Io.Write.Analog(handle, port.motor, 0)
          TRUE
            SEQ
              previous.motor.dir := speed < 0
              Io.Set.Bit(handle, port.motordir, previous.motor.dir)
              Io.Write.Analog(handle, port.motor, 2048+abs(speed))
:
```

Listing 4.13: *Elevator Example: The Panic Process*

```
PROC Filter.Panic(EVENT panic.event, no.panic.event, ...)
  CHAN SIGNAL do.panic.signal, dont.panic.signal:
  BOOL panic.state:
  PAR
    WHILE TRUE
      HANDLE panic.event
        TIME 10 MSEC -- panic event
          do.panic.signal ! signal
    WHILE TRUE
      HANDLE no.panic.event
        TIME 1 SEC -- end–of–panic event
          dont.panic.signal ! signal
    WHILE TRUE
      ALT
        do.panic.signal ? CASE
          signal
            Update.Panic(panic.state, TRUE, panic.signal, panic.light, ...)
        dont.panic.signal ? CASE
          signal
            Update.Panic(panic.state, FALSE, panic.signal, panic.light, ...)
:
```

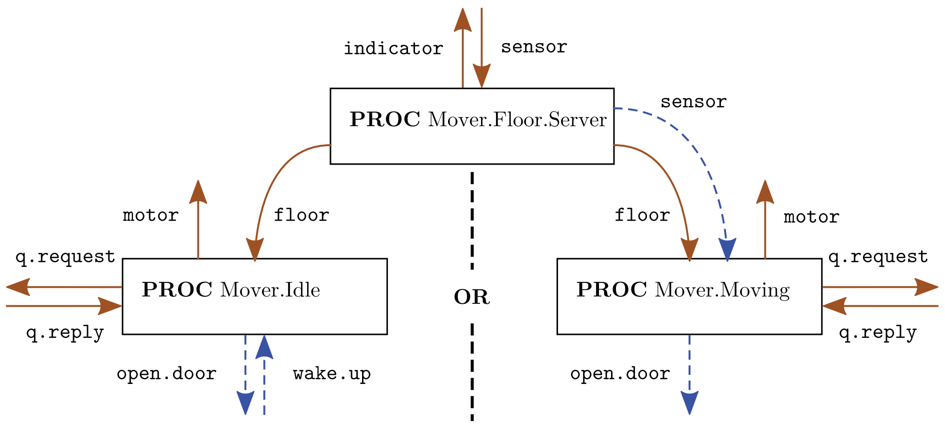


Figure 4.5: *Elevator Example: Overview of the Mover Process*

Listing 4.14: *Elevator Example: The Mover.Moving Process*

```
PROC Mover.Moving(EVENT sensor, open.door, CHAN QUEUE.REQUEST req, ...)
  HANDLE sensor
    TIME 100 MSEC −− floor sensor event
      FLOOR f:
      SEQ
        −− Request current floor from the floor server, then ask Queue what to do.
        floor ? f
        request ! what.now ; f ; dir
        reply ? CASE
          move ; dir
            motor ! dir
          sleep
            SEQ
              dir := dir.none
              motor ! dir.none
              idle := TRUE
          open
            SEQ
              motor ! dir.none
              idle := TRUE
              RAISE open.door
:
```

clients and servers can be chosen freely: For example, in listing 4.14 the Mover.Moving process performs a server call to Mover.Floor.Server with a single *input* on the floor channel.

The Mover.Idle and Mover.Moving processes do not contain main loops, in contrast to most of the other major processes in the system. Instead, they handle a single event and then return to the parent Mover process, which then decides whether to call Mover.Idle or Mover.Moving.

The Mover.Moving process uses a 100 ms deadline for handling the sensor event. When also considering the polling period, this guarantees a maximum 100+40 ms stimulus–response time from the time the elevator activates a sensor until the motor has been stopped. The Mover.Idle process is given a relaxing 1 s to restart the elevator in response to a wake.up event; this (plus 40 ms) represents the maximum allowed delay from a user presses a button, or from the door closes, until the elevator must have begun to move in response.

4.6.6 The Queue Process

The Queue process (listing 4.15) is a passive server with no temporal specifications of its own. It holds the state of the queue (which buttons that have been pressed on which floors), and contains the queuing logic that constitutes the elevator algorithm. It serves both the Mover processes, which send what.now requests to the queue; and the FilteredIO process, which sends button press updates.

On button presses, the Queue process requests that the corresponding light is lit, updates the internal queue state and, if the elevator is idle, raises a wake.up event. It also outputs a same.floor.button signal to the Door process whenever the elevator is standing still and a button request for the current floor is pressed.

On what.now requests, the Queue process checks the queue and determines what the elevator should do at this point. Because the Mover processes require a reply to the call, the call cannot be handled using an extended rendezvous, which will have to be suppressed by inserting a **SKIP** as the during process (the first process in the guard). The system will deadlock if this **SKIP** is forgotten.

Because the server call is not extended, the rendezvous is considered complete when the channel communication on the request channel has been performed, and after that the Queue process will no longer inherit the deadline of the caller. However, the Queue process will re-inherit this deadline when the calling process attempts to communicate on the reply channel, until the communication on this channel is also completed. It is therefore

Listing 4.15: *Elevator Example: The Queue Process*

```
PROC Queue(...)
  ...
  WHILE TRUE
    ALT
      ...
      request ? CASE
        what.now ; current.floor ; current.dir
          SKIP -- This SKIP suppresses extended rendezvous
          SEQ
            What.to.do(button.light, queue, current.floor, current.dir, sleeping)
            IF
              current.dir <> dir.none
                reply ! move ; current.dir
              NOT sleeping
                reply ! open
              sleeping
                reply ! sleep
:
```

little difference between the temporal behavior of a server call that uses extended rendezvous and a server call that uses explicit request and reply channels.

4.6.7 The Door Process

The main challenge of implementing the door functionality is to correctly handle the temporal specifications: the door should close 10 s after it was opened, or 10 s after the last time an obstruction was removed, or a button on the same floor was pressed, whichever is latest. This requires a **HANDLE–TIMEOUT** construct where closing the door happens on timeout. Therefore, any stimuli that should make the door not close must raise an event. In the case of an obstruction it is not so that the door should close 10 s after the obstruction event occurred; rather it should close 10 s after the obstruction is removed. This means that one must either introduce an end-of-obstruction event, or keep repeating the obstruction event for as long as the obstruction is present.

The implementation does the latter. For this, it uses an event server, called Door.Obstruction.Server (listing 4.16), which remembers the state of obstruction, and re-raises the obstruction event each time the main door process, Door.Door (listing 4.17), begins its 10 s countdown and the obstruction is still present.

Listing 4.16: *Elevator Example: The Door.Obstruction.Server Process*

```
PROC Door.Obstruction.Server(...)
  BOOL obs, watch:
  WHILE TRUE
    ALT
      same.floor.button ? CASE
        null
          If.Watch.Raise.Obstructed.Clear.Watch(obstructed, watch)
      obstruction ? obs
        IF
          obs
            If.Watch.Raise.Obstructed.Clear.Watch(obstructed, watch)
          TRUE
            SKIP
      obstruction.watch ? watch
        IF
          obs
            If.Watch.Raise.Obstructed.Clear.Watch(obstructed, watch)
          TRUE
            SKIP
:
```

Listing 4.17: *Elevator Example: The Door.Door Process*

```
PROC Door.Door(...)
  WHILE TRUE
    HANDLE open.door
      BOOL close:
      SEQ
        TIME 1 SEC -- opening the door
          SEQ
            door.light ! TRUE
            obstruction.watch ! TRUE
        WHILE NOT close
          HANDLE
            obstructed -- obstruction event
              TIME 10 MSEC
                obstruction.watch ! TRUE
            TIMEOUT 10 SEC -- keeping the door open
              TIME 1 SEC -- closing the door
                SEQ
                  close := TRUE
                  obstruction.watch ! FALSE
                  door.light ! FALSE
                  RAISE wake.up
:
```

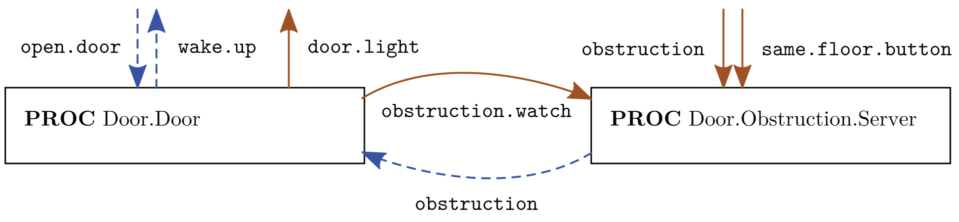


Figure 4.6: *Elevator Example: Overview of the Door Process*

To signal that it begins a new countdown, the main door process will send **TRUE** through the obstruction.watch channel. When the door is closed it will send **FALSE** through the same channel to signal that it is no longer interested in receiving obstruction events. The arrangement of the processes is illustrated in fig. 4.6.

The handling of the obstruction event itself becomes somewhat artificial due to the lack of an actual door mechanism; it involves simply not closing the door, but instead to begin another 10 s countdown.

4.6.8 Deadlock and Stall Analysis

To show that the system is free from deadlocks one can use the criteria for deadlock freedom in client–server systems (theorem 3.1).

When applied to this system, the servers are the processes that contain alternations, and the clients are processes that communicate with these alternations, although in general there may be servers without alternations if they serve only a single client. The raising of events need not be considered for deadlock analysis, as this will never cause a process to be blocked.

Theorem 3.1 contains three criteria, which if all satisfied, imply that the system is free from deadlocks:

1. The first criterion is that a client may not communicate with other processes between a server call and the reply. Most of the server calls in the elevator implementation have an implicit reply due to the use of extended rendezvous, in which case the request and reply is the same statement with no room for communication in between. The exception is the what.now call from Mover to Queue. Here one may see that the criterion holds by inspecting the source code of listing 4.14 and noting that there is no communication between the request and the reply statements.

2. The second criterion is that a server should not accept new calls when

it is already serving a call from a client. This can be verified by noting the absence of nested alternations, so that all channel communication performed during ongoing server calls is indeed calls to sub-servers where the server acts as a client.

3. The third criterion is that the client–server relation graph must be acyclic. The client–server relation graph is visualized in fig. 4.7, with the direction of arrows being from clients to servers. Note that in order to show that the graph is acyclic one has to split composite processes such as FilteredIO into sub-processes. The graph is arranged so that all clients are strictly above servers that they communicate with, and the graph is therefore acyclic.

All the criteria are satisfied, and the system must therefore be free from deadlocks.

The system is free from stalls if no process subject to a deadline constraint attempts to communicate synchronously with a process that may be under a delay constraint (observation 3.7). As can be seen in fig. 4.7, this is true for all communications except for calls to the IO.Output process, which will delay for 5 ms while stopping the motor, yet acts as a server to other processes.

All processes that directly or indirectly communicate with IO.Output therefore risk being stalled for up to 5 ms. As it is, the 5 ms delay is shorter than all other deadlines, so although other processes may stall their deadlines may still be met. (Whether or not they are actually met depends on their execution time). The current implementation of the breaking constraint is therefore not particularly good from a program quality point of view, because of the unfortunate coupling between the value of the breaking duration and the value of other deadlines in the system. The stalls could have been avoided by handling the stopping of the motor in a separate process and triggering it as an event.

4.7 Discussion

This chapter has discussed the Toc programming language, the implementation of a Toc compiler and run-time system, and their use on an example system involving control of a miniature elevator model. In this section, results from the elevator implementation will be used to evaluate the Toc language design, and the Toc lazy scheduling model.

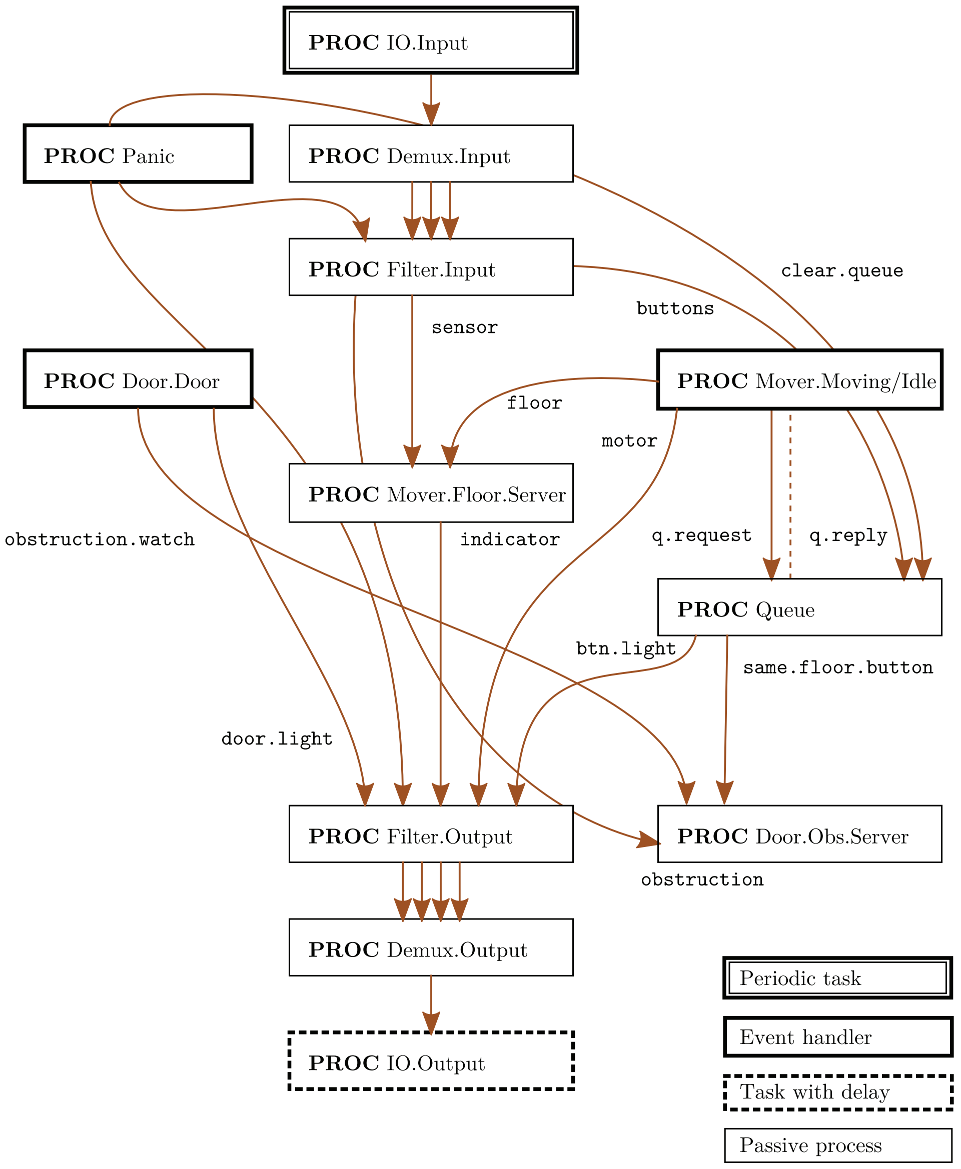


Figure 4.7: *Elevator Example: Acyclicity of the client–server relation graph*

Listing 4.18: *Possible Design of Toc Event Alternation*

```
HANDLE
  e1
    ...
  e2
    ...
  TIMEOUT
    ...
```

4.7.1 Evaluation of Toc Language Design

The elevator implementation revealed cases where the Toc language can be improved.

For example, in the FilteredIO.Panic process (listing 4.13), a somewhat complex implementation is required for a simple specification because Toc lacks support for alternation between events. Syntactically, this would be a relatively simple feature to include. A mechanism for alternation between events should be distinct from **ALT** because the latter is used in servers, and event-handling processes must never act as servers (observation 3.6). A better alternative would be to allow multiple events under a single **HANDLE**, as illustrated in listing 4.18. Such a mechanism would be similar to the handle–timeout construct of the DPS (see listing 3.8).

In FilteredIO.Panic the events are mutually exclusive, and one will not be raised while the other is being handled. When events are not mutually exclusive, then other factors must also be taken into consideration, such as how the execution time of one event handler will affect the response time of the others, and how to prioritize between multiple events that are ready to be handled. The workaround used in the elevator implementation—parallel handle constructs in parallel with a common server—is not very readable, but it does have a temporal behavior that is relatively simple to understand.

Another issue is the design of the **RAISE** and **HANDLE** primitives themselves. Their behavior was chosen so that event triggers would behave similar to counting semaphores, with one handling of an event per raising of the trigger. This has the advantage of being race condition free, and therefore easier to use as a building block for creating more complex event managing processes.

However, limiting the language to a single behavior may have been unnecessary: Even though the use of counting semaphore event triggers is the best choice for event managing processes, there is no particular reason why Toc should not directly support other type of event counters as well. One

Listing 4.19: *Entry Call Equivalent of Extended Rendezvous*

```
select
   accept Call (...) do
      During;
   end Call;
   Afterwards;
or
  ...
end select;
```

solution would be to allow event variables to be created with a maximum counter value. For example, the following syntax might have been used to create a counting semaphore trigger, a binary semaphore trigger, and a suspend–resume style trigger, respectively:

```
EVENT counting.semaphore:
EVENT(1) binary.semaphore:
EVENT(0) suspend.resume:
```

For the last type of trigger, raised events would only be handled if the handling process is currently waiting for the event, otherwise they would be discarded. This type of trigger would have simplified the obstruction handling, as one would no longer need to manually ensure that late obstruction signals from one floor is remembered when the doors open at the next floor (see listings 4.16 and 4.17).

Another design choice was to make **ALT** rendezvouses extended by default, because this seemed to be more intuitive, and because it removed a class of subtle and hard-to-find timing bugs caused by forgetting to make a communication extended when it should have been. The drawback is that communication guards that cannot be extended may now deadlock if the programmer forgets to make then non-extended by inserting a **SKIP** as the during process (see eg, listing 4.15). Either communication guard rendezvouses are extended by default, or they are not, and as neither choice is entirely satisfactory this problem cannot be fully solved. This dilemma would not have appeared if using Ada-style synchronous entry calls to communicate rather than channels. Then, all communications would be extended, and there would be a clear syntactic difference between the during process and processes to be executed afterwards, as shown in listing 4.19.

All but one server call in the elevator implementation uses extended rendezvous, so the choice of making them default was appropriate, at least for this system. Requiring the extra **SKIP** whenever a server replies to its caller through another channel is not a nice syntactic rule. Still, the deadlock that is caused by forgetting this is the type of bug that is always found during testing, as the system will never work when the **SKIP** is omitted. The subtle timing bugs that would have occurred if it was possible to program a non-extended rendezvous when an extended rendezvous was intended would have been much harder to catch.

4.7.2 Evaluation of Lazy Scheduling and Laziness Hypothesis

When the scheduler is lazy, it is possible for the system to end up in a deadlock-like state, where it is not doing anything because there are no active deadlines. Unlike a deadlock, which is caused by a circular wait, this situation is instead caused by the programmer failing to provide a deadline constraint.

An example would be if the programmer forgets to specify a deadline for initialization in the elevator implementation: the entire system would then act as **STOP** and do nothing. It may be difficult to spot these errors, as the statements involved are not executed even though they are not explicitly blocked, and even though the system is not doing anything else.

This kind of programming error would undoubtedly become common if lazy scheduled systems were to be used. A compiler could in many cases detect this type of error and issue a warning message. In some instances, such as when no **TIME** constructs are reachable from Main(), detecting the situation would be trivial. However, in other cases, such as those involving deadline inheritance through channel arrays indexed by complex expressions, detecting the situation might not be feasible. The current compiler implementation never issues this kind of warning.

This chapter began by stating the laziness hypothesis: that any part of a real-time system that cannot be given a meaningful deadline can be omitted. Because the Toc scheduler is lazy, results from the elevator implementation can be used to evaluate this hypothesis, and to assess whether or not lazy scheduling is a useful property to include in a programming language.

The primary effect of the laziness hypothesis as applied to Toc, is that programmers are required to provide deadlines for all functionality in the system. However, programmers may choose to circumvent this requirement, for example by assigning a deadline of 0 to a set of processes, which in effect will yield non-deterministic scheduling priorities (and a lot of missed deadlines). Another effect of the lazy scheduler is that it prevents the use of background tasks, ie, non-real-time tasks. Again, this can be circumvented by assigning very long deadlines to tasks. Because processes may be assigned deadlines that are nonsensical or arbitrary, the validity of the laziness hypothesis in practice, and the usefulness of a lazy scheduler, depend largely on whether all functionality can be given deadlines whose values are *meaningful*.

The temporal constraints for the elevator implementation and the associated time values are listed in table 4.3. An attempt to categorize the values of the constraints with respect to how they are determined is given below; this may give an indication of just how meaningful each actual value can be said to be.

Precise

Some constraints, such as the polling interval, have values that can be considered "physical constants" in some loose sense. It is not necessary to poll a button every millisecond, because no user would be physically able to push a button for such a short time. Moreover, if users are unable to push a button for a shorter time than, say, 40 ms, then there is no practical benefit of polling more often than that. The minimum time of a button press can therefore be considered a constant (albeit one that may be difficult to find), which makes it is possible, at least in theory, to determine precisely what the polling interval should be.

Other temporal constraints that may fall into this category are the breaking duration for the motor and the response time for the floor sensor event: The precision needed when stopping at a floor can be considered a physical property of the system, and these temporal constraints a function of this property.

Limit of acceptability

If a user pushes a panic button then the elevator should stop. If it was possible for the elevator to stop immediately, then that would be optimal, but as it is not, then some upper bound on acceptable stop times would have to be used instead. Compared to a "precise" constraint, it is harder to argue that there exists a correct answer to what the values of such a constraint should be, rather it is more of a compromise between what one would ideally want, and what the system is able to achieve.

The response-time of an obstruction event also falls into this category.

Design choice

Some temporal constraints have values that can be chosen relatively freely, and where the actual value represents a design choice of the type that would typically be part of the system specification. How long to keep the doors open at each floor is an example of this. Too short or too long, and users may experience a lower quality of service, but apart for that the exact value is a matter of choice.

Because the system is merely simulating elevator doors, the time it takes to open and to close these "doors" is also in this category.

Arbitrary

This leaves three temporal constraints: the deadline for handling the end-of-panic event, the wake up from idle time, and the deadline for initialization. Although there is a range of acceptable values—the elevator should not require an hour to wake up from idle state, for example—the range of acceptable values span many orders of magnitude and the exact value is of little importance. Moreover, the need for a separate constraint is so obscure that it is unlikely to appear in the specification of the system.

These arbitrarily valued temporal constraints deserve special notice, as their mere presence calls into question whether or not all deadlines can be said to have meaningful values.

Take the initialization deadline, for example. A millisecond could be an acceptable value for this deadline, but for a real elevator, so could a minute. The system is not doing anything else at the time of initialization, so the choice of deadline will not have any effect on scheduling. It is therefore hard to argue that initialization can be given a meaningful deadline. On the other hand, if the elevator required a year to initialize, this would obviously be too much, so there *is* a deadline somewhere in the interval between a second and a year, that if missed would be essentially the same as a failure. Still, it may be more appropriate to use **TIME** 0 or an equivalent to schedule initialization, than to provide an arbitrary valued deadline for it.

Another arbitrarily valued temporal constraint is the deadline for handling the end-of-panic event. The introduction of this event was a consequence of the choice of modular composition in the system; a choice which was made because it enabled the use of a well-established method for proving the absence of deadlocks. The more intuitive implementation of using an end-of-panic channel would not have required a separate deadline, but would have lead to a circular chain of client–server calls. The fact that design choices in the implementation may lead to changes in the set of temporal constraints is significant, and implies that for Toc programs, real-time constraints cannot be considered solely to be in the domain of the system specification.

Despite the occasional need for arbitrary deadlines, lazy scheduling has certain advantages. For one thing, it ensures completeness of the temporal specifications of a system, so that functionality that should be given a deadline must be given a deadline. In contrast, systems that use fixed priorities or contain background tasks may suffer starvation of processes when the system load is high, without this explicitly being an error. By assigning deadlines to all functionality in the system one avoids this kind of silent starvation in cases of high load.

In Toc, inconsistent specifications will be reported at run-time. For example, the floor sensor event indirectly controls the motor as part of its deadline, and may be stalled for the 5 ms delay required to stop the motor. If the programmer assumed that the floor sensor event could be handled in less than 5 ms, and assigned such a deadline to it, then this mistake would be reported run-time as “missed deadline” warnings, explicitly informing the programmer of the mistake. Systems that are scheduled using priorities cannot automatically provide this kind of feedback.

Part II

Schedulability Analysis

Chapter 5

Preliminaries on Schedulability Analysis

I love deadlines. I like the whooshing sound they make as they fly by.

D. ADAMS

QUITE DIFFERENT from the question of how to program a real-time system in order to best implement its timing requirements, is the question of, when given an already programmed real-time system, whether its timing requirements will be met. Variations in execution times and changes in the arrival patterns of sporadic tasks means that a worst-case is unlikely to be found merely by testing, instead requiring a more analytic approach.

The problem is essentially twofold. One part is the problem of determining the worst-case execution times (WCETs) of the different components of the system when executing on the chosen computing platform; that is, the mapping from the source code of a single process to some upper bound on its execution time.

The second part is the schedulability analysis itself, which combines these WCETs to determine whether processes meet their deadlines when being executed together on the same system. This is the problem discussed in this part of the thesis.

Both WCET and schedulability analysis requires adherence to certain constraints. WCET analysis, for example, typically prohibits features with unpredictable or non-deterministic execution times, such as recursion, unbounded (**while**) loops and dynamic memory.

Schedulability analysis imposes other kinds of limitations. Some analysis techniques assume implicit deadlines; others require that only mutual exclusion synchronization is used to communicate between processes, or that a specific synchronization protocol must be applied. For multiprocessor systems, a common requirement is that each job must in itself be serial and contain no parallel processes.

Principles of process-oriented design are in direct conflict with two of these requirements:

1. Process-oriented design prohibits mutual exclusion synchronization, while most schedulability analyses assume it.

2. In process-oriented programs, parallel processes are used as a means of structuring and organizing a program. Most schedulability disallows this by requiring that each deadline constraint maps to exactly one thread of execution.

This part of the thesis will present three contributions to schedulability analysis, aimed at making the analysis less incompatible with process-oriented design. First, in chapter 6, a schedulability analysis for synchronously communicating systems is presented. The communication between processes must have a client-server structure. The analysis supports both fixed priority scheduling (FPS) and earliest deadline first (EDF), but only works for uniprocessor systems.

The second two contributions relate to multiprocessor schedulability analysis. Chapter 7 presents a formal model and algebra for reasoning on the temporal behavior of real-time processes with a complex parallel structure. The model assumes discrete time, and an intra-job scheduler that is work-conserving, but otherwise undefined. The analysis provides fundamental insight into the temporal behavior of parallel jobs when executing on multiple processors.

In chapter 8, a schedulability analysis framework is presented for systems of independent real-time processes where jobs are allowed a complex parallel structure. In contrast to the analysis of chapter 7, this analysis assumes continuous time, and an intra-job scheduler that is reasonably fair.

5.1 Basics, Terminology and Nomenclature

Moving to this second part of the thesis will inevitably involve some change in terminology, as the topic turns away from implementation of real-time processes towards an abstract representation of their execution times. This section will introduce the basic terminology; a complete nomenclature for this part of the thesis is given in appendix C. Also refer to the general introduction to real-time scheduling in section 2.4 for other relevant terms and definitions, as these will not be repeated.

It will be assumed that the systems to be analyzed can be modeled as a set of cyclic, top-level tasks that execute in parallel, and unless otherwise noted, independently of other tasks. A task is real-time if it is subject to temporal constraints, and non-real-time otherwise. Tasks will be denoted with upper-case letters $(A, B, \dots)$.

Each instance of a real-time task is called a job and will be denoted with lower-case letters $(a, b, \dots)$. A job has a release, denoted r, and an absolute deadline for when it must have completed its work, which will be denoted d. Tasks have relative deadlines, denoted D, which specifies the time between the release and deadline for jobs from that task $(r + D = d)$. The period of a task A will be denoted T_A, and is defined to be the minimum inter-arrival time (MIT) of jobs, even for tasks that are periodic; this provides a convenient unified way of modeling both sporadic and periodic tasks.

If FPS is used then all tasks must be given a fixed priority, which will be denoted P_A for each task A. The effective scheduling priority of a task A will be denoted $\pi(A)$. Under EDF the priority will change for each job. The effective priority may also be changed by the synchronization protocol, and so may not be constant even under FPS.

For all priority driven schedulers, each task A can be assigned a preemption level, denoted $\widehat{\pi}(A)$. If $\widehat{\pi}(A) \geq \widehat{\pi}(B)$, also written $B \prec A$, then B will never preempt A; that is, it will never be the case that a newly released job of B has a higher priority than an existing job of A. Under FPS, the preemption level of a task is equivalent with its priority. Under EDF, $B \prec A$ is equivalent to $D_A \leq D_B$.

For systems where all jobs are serial, the WCET of a job or task can be described by a single scalar, which is denoted C. The utilization of a task A, denoted U_A, is its computational requirement divided by its period, or C_A/T_A. The utilization U_{sum} of a system is the sum of the utilizations of its tasks:

$$U_{\text{sum}} = \sum_{X \in \mathbb{T}} U_X = \sum_{X \in \mathbb{T}} \frac{C_X}{T_X} \tag{5.1}$$

where $\mathbb{T}$ is the set of tasks.

It is sometimes convenient to define the density of a job A, denoted λ_A, as its computation divided by its relative deadline, and the density of a system, λ_{sum}, as the sum of the density of its tasks:

$$\lambda_{\text{sum}} = \sum_{X \in \mathbb{T}} \lambda_X = \sum_{X \in \mathbb{T}} \frac{C_X}{D_X} \tag{5.2}$$

WCETs are always upper bounds, and a schedulability test must therefore take into account that tasks may execute less than their WCETs. Similarly, only a lower bound on inter-arrival time is known for sporadic tasks. A schedulability analysis is sustainable [9] if a system determined to be schedulable remains schedulable when decreasing its required work, increasing task periods or in other ways relaxing its timing requirements. A schedulability analysis is called exact if it provides sufficient and necessary conditions for schedulability; or pessimistic if it only provides sufficient conditions.

5.2 Uniprocessor Schedulability Analysis

Real-time schedulability analysis for uniprocessor systems is a well-developed field. Most analyses have a basic form for task sets that are independent, where tasks never share resources or in other ways block the progress of other tasks. These analyses are then extended to the more general case of communicating tasks.

5.2.1 Independent Task Systems

Systems of independent, implicit deadline tasks are often called Liu-Layland (LL)-systems, because of two famous schedulability tests by Liu and Layland [75]. The first of these applies to systems scheduled with EDF, and illustrates the optimality of the EDF scheduler on multiprocessor systems:

Theorem 5.1 (LL-EDF [75]). *An EDF-scheduled LL-system is schedulable, if and only if*

$$U_{\text{sum}} \leq 1 \tag{5.3}$$

The second test gives sufficient, but not necessary conditions for a system to be schedulable under a fixed-priority scheduler, when priorities are ordered by decreasing periods (RMPO):

Theorem 5.2 (LL-RMPO [75]). *An RMPO-scheduled LL-system is schedulable if*

$$U_{\text{sum}} \leq N\left(2^{1/N} - 1\right) \tag{5.4}$$

where $N = |\mathbb{T}|$ is the number of tasks.

The right hand side of eq. (5.4) converges to $\ln 2 \approx 0.69$ for large values of N, so any LL-system with utilization lower than this is schedulable under

RMPO. However, the bound given in eq. (5.4) is pessimistic, and systems with higher utilizations may be schedulable in practice. A more recent and better test for this type of tasks is the hyperbolic bound test [18], which accepts a much larger ratio of schedulable task sets.

There are a number of problems with utilization-based schedulability tests: They must assume implicit deadlines; they do not work for arbitrary priority orderings and they are not exact when applied to fixed priority systems.

An alternative is response-time analysis (RTA) [58], which provides an exact schedulability test for fixed-priority systems of any priority ordering and for constrained deadlines ($D \leq T$). As the name implies, RTA works by computing the worst-case response time of a task, which for task A is denoted R_A, and is the maximum time between release and completion for any job of A. When $R_A \leq D_A$, then A will meet its deadlines. For fixed-priority systems of independent tasks, the response-time of a task is its own execution time plus the sum of interference from higher priority tasks. This is a recursive problem, as the number of higher priority jobs that may interfere again depends on the response-time:

Theorem 5.3 (RTA [58]). *A constrained deadline system is schedulable under* FPS *if, for all tasks $A \in \mathbb{T}$*

$$R_A \leq D_A \tag{5.5}$$

where

$$R_A = C_A + \sum_{X \in \mathbb{T}:\ X > A} \left\lceil \frac{R_A}{T_X} \right\rceil C_X \tag{5.6}$$

Equation (5.6) can be solved as a fixed-point equation, by iterating from the starting point $R_A = C_A$ [1].

RTA cannot easily be used with EDF. An analysis method that works with EDF for independent task sets, and for arbitrary deadlines, is the processor demand criterion (PDC) [10, 11]. The PDC is based on the observation that for a system to be unschedulable under EDF, there must exist some window in time where jobs with release and deadline within the window require more computation than is available in that window. The worst-case required computation as a function of window length t is called the demand-bound function, and is denoted DBF(t). The worst-case demand-bound of any window can found by assuming that all tasks are released synchronously and at maximum rate, and then computing the sum of their demands:

$$\text{DBF}(t) = \sum_{X \in \mathbb{T}} \max \left\{ 0,\ \left(1 + \left\lfloor \frac{t - D_X}{T_X} \right\rfloor \right) C_X \right\} \tag{5.7}$$

The downward saturation to zero is only required for non-constrained deadline tasks; ie, when $D > T$.

The PDC can then be stated as follows:

Theorem 5.4 (PDC [10, 11, 109]). *A task system is schedulable if*

$$\forall l \in \mathbb{R}^+ : \textsc{dbf}(l) \leq l \tag{5.8}$$

Two useful upper bounds l_{ub} are known such that if theorem 5.4 holds for all l up to either of these bounds, then it will hold for any l. One upper bound is based on the utilization of the system [109]:

$$l_{\text{ub}} = \max\left\{\max_{X\in\mathbb{T}}(D_X - T_X),\ \frac{1}{1-U_{\text{sum}}}\sum_{A\in\mathbb{T}} U_A \cdot (T_A - D_A)\right\} \tag{5.9}$$

another on the length of the synchronous busy period, which is the period from 0 to the first processor idle time, given that all tasks are released simultaneously and at their maximum rates [87, 94]. The value of this upper bound can be found by the (guaranteed) convergence of the recursion

$$w_0 = \sum_{A\in\mathbb{T}} C_A \qquad\qquad w_{n+1} = \sum_{A\in\mathbb{T}} \left\lceil \frac{w_n}{T_A} \right\rceil C_A \tag{5.10}$$

that is, the upper bound l_{ub} is equal to any w_n for which $w_{n+1} = w_n$.

These two alternative upper bounds are independent, and the two are combined in the Quick processor-demand analysis (QPA), by Zhang and Burns [109], which is a PDC-based schedulability test that is much faster than previously existing tests.

5.2.2 Mutual Exclusion Synchronization

Traditional schedulability analysis typically support synchronization between tasks using resources shared under mutual exclusion. Because of the problem of unbounded priority inversions, it is necessary to apply a synchronization protocol (see section 2.4) in order to analyze systems of communicating tasks. These protocols make it possible to provide upper bounds on the maximum interference experienced by a task due to blocking by lower priority tasks. This can then be incorporated into a schedulability analysis. For fixed priority systems, RTA can be modified to account for blocking by simply adding a blocking term to eq. (5.6):

Theorem 5.5 (RTA with blocking [28]). *A constrained deadline system is schedulable under* FPS *if, for all tasks $A \in \mathbb{T}$*

$$R_A \leq D_A \tag{5.11}$$

where

$$R_A = C_A + B_A + \sum_{X \in \mathbb{T}:\, X > A} \left\lceil \frac{R_A}{T_X} \right\rceil C_X \tag{5.12}$$

and B_A denotes the maximum time a job from task A may be blocked waiting for lower priority tasks.

A task may be blocked directly, by sharing a resource with a lower priority task; or indirectly, if some higher priority task shares a resource with a lower priority task. The *usage function* is useful for describing where blocking can occur [28]. If $\mathsf{shared}(A, B)$ denotes the set of resources shared between tasks A and B, the usage function can be defined as follows:

$$\mathsf{usage}(r, A) = \begin{cases} 1 & \text{if } \exists L, H \in \mathbb{T} \text{ s.t. } L < A \leq H \wedge r \in \mathsf{shared}(L, H) \\ 0 & \text{otherwise} \end{cases} \tag{5.13}$$

which reads: if r is a resource and A a task, $\mathsf{usage}(r, A)$ is 1 if it is possible that task A may, directly or indirectly, be blocked due to the sharing of r, and 0 otherwise.

If a system uses the priority inheritance protocol (PIP), a worst-case situation can be assumed where all shared resources are held by lower priority tasks. The maximum blocking B_A for a task A is then

$$B_A = \sum_{r \in \mathbb{S}} \mathsf{usage}(r, A)\, C_r \tag{5.14}$$

where C_r is the WCET of the critical section of resource r, and $\mathbb{S}$ is the set of resources in the system. If a system uses the stack resource policy (SRP) or the priority ceiling protocol (PCP), each instance of a task may only be blocked by one lower priority task, so maximum blocking can be expressed as

$$B_A = \max_{r \in \mathbb{S}} \mathsf{usage}(r, A)\, C_r \tag{5.15}$$

which is clearly less than or equal to the maximum blocking with inheritance.

The PDC can also be changed to account for blocking. Instead of a blocking term for each task, this requires a blocking function $B(l)$:

Theorem 5.6 (PDC with blocking [6]). *A task system scheduled under* EDF *is schedulable, if*

$$\mathrm{DBF}(l) + B(l) \le l \tag{5.16}$$

for all $l \ge 0$*, where* $B(l)$ *is the maximum time tasks with deadlines* $D > l$ *may block a task with deadline* $D \le l$*.*

An EDF usage function analogous to the fixed-priority case can be defined as

$$\mathrm{usage}_{\mathrm{EDF}}(r, l) = \begin{cases} 1 & \text{if } \exists L, H \in \mathbb{T} \text{ s.t. } D_L > l \ge D_H \wedge r \in \mathrm{shared}(L, H) \\ 0 & \text{otherwise} \end{cases} \tag{5.17}$$

If a system uses the PIP, a worst-case situation can be assumed where all resources are held by tasks with relative deadline $D > l$. In that case, maximum blocking is

$$B(l) = \sum_{r \in \mathbb{S}} \mathrm{usage}_{\mathrm{EDF}}(r, l)\, C_r \tag{5.18}$$

If a system uses the SRP, only one task with $D > l$ may be blocking another task with $D \le l$ [2]. Therefore, the blocking term is simply

$$B(l) = \max_{r \in \mathbb{S}} \mathrm{usage}_{\mathrm{EDF}}(r, l)\, C_r \tag{5.19}$$

More precise blocking terms for the PIP, which are valid when resource usage is strictly nested, will be developed in section 6.4. An example of their use is presented in section 6.5.1.

5.3 Multiprocessor Schedulability Analysis

There are two fundamentally different approaches to multiprocessor scheduling. The first is partitioned scheduling, where each task is permanently assigned to a single processor. This simplifies analysis for each processor, but introduces an optimal partitioning problem that is NP-hard [36]. The second approach is global scheduling, where tasks are allowed to migrate between processors at run-time. Here, there is no partitioning problem, but the analysis itself becomes more difficult. The two approaches are incomparable under priority driven schedulers: Some systems may only be schedulable with a partitioned approach, while other systems may only be schedulable with a global approach [7].

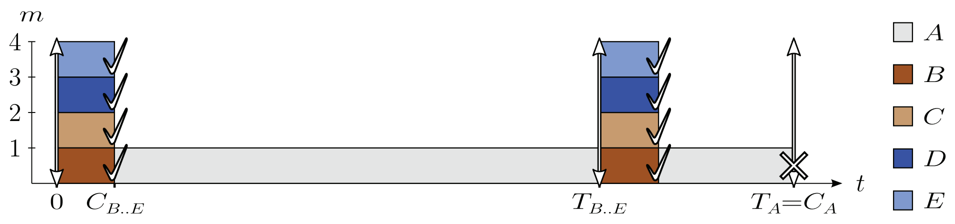


Figure 5.1: Dhall's Effect: The task set has utilization $U \approx 1$ (if $C_{B,\dots,E}$ is small), but is not schedulable under EDF even if the system contains 4 processors.

Global multiprocessor schedulability analysis is considerably more complex than uniprocessor analysis. Two well-known observations illustrate this:

1. Dhall's effect, which describes the existence of multiprocessor systems that are unschedulable under EDF on a large number of processors, even if their utilization is only 1 (ie, 100% of one processor).
2. The multiprocessor anomaly, which is a common name for the observation that in multiprocessor systems, tasks executing at their maximum rate does not necessarily constitute a worst-case scenario.

Examples will be given.

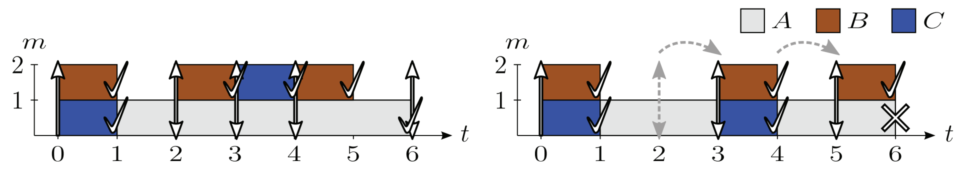
Example 5.3.1 (Dhall's Effect)
Consider a system of five tasks, $A, B, \dots, E$, executing on a system of 4 processors. Assume implicit deadlines. Let

$$C_{B,\dots,E} \approx 0^+ \qquad C_A = T_A$$
$$T_{B,\dots,E} = x \qquad \text{where } x < T_A$$

The utilization of the system is close to 1. However, task A has no slack, and consistently requires one processor in order to meet its deadline. Under EDF, however, it will be considered the least urgent, because its deadline is the latest. The scheduler will prioritize the other tasks and A will miss its deadline. This is illustrated in fig. 5.1. Note that it is possible to construct a similar system for an arbitrary number of processors.

Example 5.3.2 (Multiprocessor anomaly [example from 7])
Consider a dual processor system with tasks A, B and C, where

$$C_A = 1 \qquad C_B = 1 \qquad C_C = 5$$
$$T_A = 2 \qquad T_B = 3 \qquad T_C = 6$$



(a) (b)

Figure 5.2: The multiprocessor anomaly. Task set is schedulable when executing at maximum rate (a), but not when a task release is delayed (b).

These tasks are schedulable if released synchronously and at their maximum rates (fig. 5.2a), but not if the second job of A is delayed by 1 (fig. 5.2b). This is because C only suffers interference when A and B executes simultaneously. If $T_A = 3$ then this happens twice within each job of C; if $T_A = 2$ then it only happens once.

5.3.1 Independent Task Systems on Multiprocessors

It is possible to create a utilization/density-based schedulability test for multiprocessor systems that take Dhall's effect into account, by explicitly considering the maximum utilization or density of any task in the system. This is the approach used by the GFB-test:

Theorem 5.7 (GFB-test [17, 48]). *A constrained deadline system of independent tasks is schedulable under multiprocessor-*EDF *if*

$$\lambda_{\text{sum}} \leq m \cdot (1 - \lambda_{\max}) + \lambda_{\max} \tag{5.20}$$

where

$$\lambda_{\max} = \max_{X \in \mathbb{T}} \frac{C_X}{T_X} \tag{5.21}$$

Two other well-known schedulability analyses for multiprocessor EDF are the BAK-test [3, 4] and the BCL-test [16, 17]. A comparison in Baruah and Baker [8] found the two tests to be similar in performance, despite the BAK-test being significantly more sophisticated. The same paper presented a new test for multiprocessor EDF:

Theorem 5.8 (from [8]). *A system is global-*EDF*-schedulable upon a platform of m identical unit-capacity processors if*

$$\Lambda \leq \mu - (\lceil \mu \rceil - 1)\, \delta_{\max} \tag{5.22}$$

where

$$\Lambda = \max_{t \in \mathbb{R}^+} \left(\frac{\mathrm{DBF}(t)}{t} \right) \qquad \mu = m - (m-1)\,\delta_{\max}$$

Multiprocessor RTA-based analyses for FPS and EDF were developed by Bertogna and Cirinei [15]. The analysis for EDF has better performance than the BAK, BCL and GFB-tests, and being based on response-times, yield additional information compared with a simple yes/no to schedulability. The equations are somewhat complex and will not be re-stated here.

For uniprocessor systems of independent tasks, EDF is known to be an optimal scheduling algorithm, in that every schedulable system is schedulable under EDF. For multiprocessor systems there is no priority driven scheduling algorithm that has this property. However, a different kind of algorithm known as proportional-fair (PFAIR) [12] is known to be optimal for implicit deadline systems.

The PFAIR algorithm is an idealization of a scheduling algorithm, that for any time window of length Δt, allows each task T to progress by $\Delta t \cdot U_T$. The idea is to execute each task as slow as possible, so that each task finishes exactly at its deadline. In practice, an implementation must execute larger time slices of each task in order to avoid excessive task switches, which would reduce performance. Efficient implementations and extensions to the PFAIR algorithm are discussed in Srinivasan [95].

5.3.2 Mutual Exclusion Synchronization on Multiprocessors

In a uniprocessor platform, when a high priority job requires a resource held by a lower priority job, the system can execute the lower priority job until the resource is freed. Although the higher priority job is blocked, the system is kept busy, and no lower priority jobs will be able to lock further resources until all higher priority jobs have completed.

Neither of this applies to multiprocessor systems. If the blocking, low priority job is already executing on a different processor, the higher priority job can do nothing but wait, leaving its processor idle. Allowing other tasks to use that processor will not improve worst-case schedulability, as the highest priority job will still be blocked.

Moreover, as lower priority tasks may execute simultaneously with higher priority tasks, a high priority task can be blocked by lower priority tasks that were released after it, and if the high priority task accesses a resource several times, it may be blocked at each access, not only once, as is the case for uniprocessor systems. Furthermore, using for example the SRP in its original form will mean that when a resource with the highest ceiling is held by any task, no new jobs may start to execute on any processor, which would be inefficient.

There is no perfect solution to these problems; multiprocessor synchronization protocols must balance the problem of excessive blocking that happens when low- and high priority tasks execute simultaneously; with the problem of low utilization that happens when too many lower priority tasks are prevented from executing when processors are idle.

The first multiprocessor synchronization protocol was the multiprocessor PCP (MPCP), developed by Rajkumar et al. [84]. A similar protocol, the multiprocessor SRP (MSRP) was developed by Gai et al. [44]. Both the MPCP and MSRP require partitioned schedulers. Their theoretical performances are similar, but the MPCP behaves slightly better in practice due to less overhead [44].

These protocols divide resources into those that are local, (ie, used only by tasks executing on one processor), and those that are global (ie, shared between tasks on different processors). Uniprocessor synchronization protocols are used for local resources. For the MPCP, a task becomes non-preemptible when it acquires a global resource; for the MSRP it gains a higher preemption level than any task, preventing any new job from starting. When a task is blocked waiting for a global resource, the MPCP will block the task, allowing other tasks to execute; while the MSRP will spin the task, keeping the processor busy while it is waiting.

The first protocol for global scheduling that also allowed schedulability analysis was the EDF-hybrid (EDF-H) protocol developed by Devi et al. [38]. The EDF-H protocol is an EDF scheduler that disables preemptions for any task that holds a shared resource. Because of this it becomes essential to keep critical sections short, as all assigned scheduling priorities are subverted whenever any task is holding a resource.

A protocol called parallel PCP (P-PCP) was introduced in Easwaran and Anderson [40]. It only supports FPS and does not support nested resources. It works by allowing some tasks, limited by a parameter vector, to execute even when a strict PCP would prevent it. The parameter vector can be tuned to increase concurrent execution, or conversely, to reduce worst-case blocking.

5.3.3 Job-level parallelism (JLP)

Most existing multiprocessor analyses assume that each job is serial and is never able to simultaneously execute on multiple processors. A real-time system where one job may simultaneously utilize more than one processor is said to allow job-level parallelism (JLP). JLP implies a global scheduler.

A classification of systems with JLP is given in Goossens and Berten [47]. Here, a parallel job is defined to be rigid, if the number of processors required by the job is determined a-priori, moldable if it is determined by the scheduler, but does not change during the execution of the job, or malleable if the number of processors assigned to the job may change during the execution of the job. A task is defined to be rigid if all its jobs are rigid, and so on. Schedulability analyses of rigid tasks can be found in eg, Goossens and Berten [47] and Kato and Ishikawa [59].

A limitation with existing analysis that allow non-rigid tasks is that they do not properly take into account the parallel structure of the task that a job implements: The analysis given in Han and Lin [51] assumes that all jobs are fully parallelizable to an arbitrary degree with no overhead. The analysis in Collette et al. [34] allows costs to be added to the parallelization of a job, which can also be used to model an effective limit to the degree of parallelization, but it does not discuss how to model actual programs, or which programs that can be modeled. The analysis in Lakshmanan et al. [66] assumes that jobs have a basic fork-join structure, where the main thread of a job may fork into multiple branches, but where the branches themselves are not allowed to fork. A similar constraint is used in Berten et al. [14], where jobs with this structure are called multi-phase jobs.

Job-level parallelism is the main topic of chapters 7 and 8.

Chapter 6

Analysis of Client-Server Structures

Buffalo buffalo that Buffalo buffalo
buffalo, buffalo Buffalo buffalo.

W. J. Rapaport

In this chapter, a schedulability analysis for systems using synchronous client–server based communication will be developed. The analysis can be used on Toc programs if they have a simple parallel structure and satisfy the criteria for deadlock-freedom given in theorem 3.1. The analysis can also be used on other synchronously communicating client–server systems. It supports both EDF and FPS, and versions of the PIP and SRP protocols. It is limited to uniprocessor systems.

6.1 Introduction

When tasks communicate via shared variables and only use mutual exclusion for synchronization, then all execution that is required to complete a task will be local to the thread of execution associated with that task. Some of this code represents resource usage; its execution must not be interleaved with execution of another task accessing the same resource. In a client–server system, however, resource usage is not incorporated into the local code of a task. Instead, the resources (ie, the servers) are themselves tasks.

An intuitive first step for analyzing the schedulability of client–server systems is to consider a server shared between two tasks to be a resource shared under mutual exclusion between those two tasks. From this starting point, two significant differences from mutual exclusion synchronization become apparent:

1. A server task may execute code that is not directly part of a call from a client, and thus a server may not immediately be ready to serve a client even if no other clients currently hold the server. This is in contrast to resources in shared-memory based systems, where a resource is always ready unless another task is holding it.

2. It is no longer reasonable to assume that resource accesses can be kept short, as a significant part of the computation required to complete a real-time task may be performed as part of calls to servers.

The last point makes the extension of existing multiprocessor analysis (see section 5.3.2) such as the EDF-H more difficult, because the EDF-H relies on making a task non-preemptible whenever it holds a shared resources. The P-PCP is not suitable either, as it does not support nested resources, which are required in all practical client–server systems. The analysis presented here will therefore focus on uniprocessor systems.

In earlier literature, tasks that communicate synchronously have been considered hard to analyze [eg, 29]. The reasoning has often been the difficulty of resolving situations that arise when one job requires synchronization with another job that has not yet been released (ie, a stall). In effect, this is the problem described in observation 3.7, where it was noted that a process with a deadline should not require synchronization with a process that might be subject to a delay constraint.

However, this is only a limitation when direct synchronization between real-time tasks is actually required. More often than not, the purpose of the synchronous communication is communication, and not synchronization. Direct synchronization between real-time tasks is then easily avoided by letting the tasks communicate through an intermediate, non-real-time task. When using the client–server model for communication, this is equivalent with requiring that no real-time task may act as a server.

6.1.1 Related Work and Motivation

A schedulability analysis with support for synchronous communication in Ada was developed by Burns and Wellings [26], by the use of an abstraction called a session. In this model, tasks may only communicate synchronously with other tasks in the same session, and all tasks within the same session are required to have the same release time.

An analysis for systems designed using UML-RT is given in Saksena and Karvelas [89]. The system model is based on capsules, events and actions. Capsules are encapsulated objects with their own thread of execution, which communicate either by raising asynchronous events that triggers actions in other capsules, or by performing synchronous calls to other capsules. An event may also be from an external source or from a timer, thus allowing both periodic and sporadic tasks. The schedulability analysis yields upper bounds on the duration from an external event to the completion of each action triggered directly or indirectly by the event. Synchronization is always between capsules, and not between real-time tasks, which guarantees that no task requires synchronization with a task that has not yet been released.

In contrast to the session model of Burns and Wellings, the model in this chapter allows synchronous communication between tasks of different periods, but the communication must happen indirectly through a non-real-time server. The model does not allow the explicit synchronization possible with the session model.

Unlike the UML-RT-model of Saksena and Karvelas, servers will be allowed to execute code not part of a call from a client. However, the model in this chapter has no direct equivalent to the UML-RT asynchronous event, although this can be partially mitigated by asynchronously triggering the release of a sporadic task. The UML-RT-model also supports overlapping task instances, which this model does not.

However, as the UML-RT-model lacks support for EDF, inheritance and active execution by servers, it is unsuitable for analyzing a large set of systems that use synchronous communication, including systems that use the language primitives developed in chapter 3. A new analysis technique is therefore required.

6.1.2 Outline

The system and program models are defined in section 6.2. Also given are some constraints on communication that are required to ensure freedom from stalls and deadlocks.

Section 6.3 defines two alternative synchronization protocols: one inheritance protocol and one ceiling protocol based on the SRP. The inheritance protocol is the simplest in design and implementation, but the ceiling protocol has better performance.

The main schedulability analysis is given in section 6.4. A definition of demand is given, and blocking terms are developed for the two synchronization protocols. Then the analysis for a complete system is developed. If the scheduler is based on fixed priorities, then RTA is used; if the scheduler is based on EDF then the PDC is used.

Section 6.5 shows examples of uses of the schedulability analysis. The first is an example of applying the analysis to protected objects and the

PIP; it is shown that the blocking term derived in this paper improves on the term derived by existing methods. The second example is the use of deferred server calls, ie, server calls that delay computation generated by the call until after the call. This is a pattern that is hard to express when not using a process-oriented client–server paradigm. It is demonstrated that there exists systems that can only be scheduled when tasks communicate using deferred server calls, and not if using a protected object; this shows that synchronous communication can in some cases improve schedulability.

6.2 System Model

This section will introduce the system model used in this chapter. Although the client–server framework is flexible, there are certain rules for communication that must be followed for the analysis to work. The systems themselves are described using a simple language where statements either take up some processing time or make calls to servers, and this part of the model is introduced next. Finally, some extra notation is given that is necessary in order to analyze the systems.

It is assumed that systems consist of a fixed number of real-time tasks and servers. Real-time tasks are assumed to have constrained deadlines. Servers, on the other hand, cannot have associated timing requirements, and must always be ready either to execute or to communicate. An ideal platform model will also be assumed, with no overhead and no cost of preemption.

6.2.1 Constraints on Communication

Communication must satisfy four rules, the first three which comply with the criteria for deadlock-free client–server systems stated by theorem 3.1 [78, 105]:

Rule 6.1: *A client that has sent a request to a server must always be ready to accept the reply from that server.*

Rule 6.2: *Between accepting a request from a client and the corresponding reply, a server may not accept requests from other clients, but may act as a client to other servers.*

Rule 6.3: *The client–server relation graph must be acyclic.*

Rule 6.1 is stricter than its equivalent in theorem 3.1, which only requires that the client does not communicate with other servers during an ongoing server call. Rule 6.1 also implies that each task may hold at most one server at a time, although this server may again hold one sub-server, and so on.

Rule 6.4: *No real-time task may act as a server.*

This rule ensures that no task is blocked waiting for a job that has not yet been released, thus avoiding the problems of stalls caused by direct synchronization between real-time tasks (observation 3.7). It also prevents servers from having their own deadline constraints, which according to observation 3.6 would hinder the development of a synchronization protocol.

6.2.2 Program Model

The program model consists of a simple language where each basic statement either takes up some processor time, or makes a call to a server.

A server may include code for initialization or cleanup or post-processing after a previous call. Such code needs to be executed before the server is ready to accept a new call, but is not executed as part of such a call. When a server is executing this kind of code, it is said to be in the *request phase.* While a server is handling an accepted a call, it is said to be in the *reply phase.*

Formally, the program model is as follows:

A statement is an element of the set $\mathbb{I}$, and is either of the following operations:

$$\begin{aligned} i \in \mathbb{I} \iff & i = \text{Exec } r && \text{where } r \in \mathbb{R}^+ \\ \vee\ & i = \text{Call } S.c && \text{where } S \in \mathbb{S}, c \in \Sigma_S \end{aligned} \qquad (6.1)$$

where $\mathbb{S}$ is the set of server tasks and Σ_S is the signature of server S (see below).

An Exec r statement is an abstract statement that requires r amount of processing time to complete. It is fully preemptible; if executing for some time $r' < r$, the remaining computation can be modeled as

$$\text{Exec } (r - r')$$

The Call-statement is a synchronous call to a server, and will not complete until the server has completed the associated reply phase.

A block is a sequence of statements. Angle brackets are used to describe such a sequence, eg, $\langle \text{Exec } 10, \text{Call } S.c\rangle$. An empty block is denoted $\langle\rangle$; concatenation of two blocks b_1 and b_2 (execute b_2 after b_1) is written $b_1 \frown b_2$.

A real-time task A is modeled by its relative deadline D_A, its period T_A, and, under FPS, a priority P_A. In addition, each task has a defined set of job blocks I_A. Each job of the task executes one block, which for modeling purposes is assumed to be chosen non-deterministically. The set of real-time tasks $\mathbb{T}_{\text{RT}}$ is a subset of the set of tasks: $\mathbb{T}_{\text{RT}} \subseteq \mathbb{T}$.

An accept $S.c$ represents the call of type c to server S. Each accept has a set of code blocks $I_{S.c}$ that handle a server call of that type. For each accepted call, one non-deterministically chosen block is executed. The full set of accepts for a server S (the signature of S) is denoted Σ_S. For completeness, the signature of a real-time task is defined to be the empty set.

A server task S has a set of request phase blocks I_S, one of which must be executed before each accepted call, and a set of accepts Σ_S. The set of server tasks $\mathbb{S}$ is a subset of the set of tasks: $\mathbb{S} \subseteq \mathbb{T}$.

6.2.3 Program Examples

Two examples of using the program model will be given; the first is an example of a server written in Ada, the second of a real-time task written in Toc.

Example 6.2.1 (Modeling a Server)
Let A be the server described by the Ada pseudo-code in listing 6.1. Let C_p be the WCET of evaluating the expression 'pred', and $C_1, \ldots, C_6$ be the WCET of the statements S1,...,S6, all of which are assumed not to contain further communications.

When accepting a call of type Call_a, the server executes a simple block S2, so

$$I_{A.a} = \{\langle \text{Exec } C_2 \rangle\}$$

When accepting a Call_b, the handing depends on the value of pred, so the set $I_{A.b}$ has two different blocks:

$$\begin{aligned} I_{A.b} = \{&\langle \text{Exec } C_3, \text{Call } B.a \rangle, \\ &\langle \text{Exec } C_5 \rangle\} \end{aligned}$$

The signature Σ_S of S is the set of accepts:

$$\Sigma_A = \{(a, I_{A.a}), (b, I_{A.b})\}$$

Listing 6.1: *Ada Server Pseudo-code for Example 6.2.1*

```
task body Server_A is
begin
    S1 (...);
    loop
        select
            accept Call_a (...) do
                S2 (...);
            end Call_a;
            Server_C.Call_a(...)
        or
            when pred =>
                accept Call_b (...) do
                    S3 (...);
                    Server_B.Call_a(...);
                end Call_b;
                S4 (...);
        or
            when not pred =>
                accept Call_b (...) do
                    S5 (...);
                end Call_b;
        end select;
    S6 (...);
end Server_A;
```

Listing 6.2: *Toc Task Pseudo-code for Example 6.2.2*

```
PROC Task.B(...)
  WHILE TRUE
    TIME 100 MSEC
      TIME 20 MSEC
        SEQ
          P1(...)
          server.S.call.a.request ! ...
          server.S.call.a.reply ? ...
          P2(...)
          IF
            pred(...)
              server.S.call.b ! ...
            TRUE
              P3(...)
:
```

Before the first call, S5 must be executed and pred must be evaluated. The code to execute between subsequent calls depends on the last call. The common component is S6 and the re-evaluation of pred, but with an additional S4 or a call to Server_C. These alternatives constitute the request phase blocks of the server:

$$\begin{aligned} I_A = \{ &\langle \text{Exec } C_1, \text{Exec } C_p \rangle, \\ &\langle \text{Call } C.a, \text{Exec } C_6, \text{Exec } C_p \rangle, \\ &\langle \text{Exec } C_4, \text{Exec } C_6, \text{Exec } C_p \rangle, \\ &\langle \text{Exec } C_6, \text{Exec } C_p \rangle \} \end{aligned}$$

The complete server is modeled by the pair (I_A, Σ_A).

Example 6.2.2 (Modeling a Real-time Task)
Let B be the real-time task described by the Toc pseudo-code in listing 6.2. Let C_p be the WCET of evaluating pred(), and $C_1, \ldots, C_3$ be the WCET of the statements P1(),...,P3(), which contain no communications.

Task B contains two execution paths that vary in their communications. If b_1 is the sequence of instructions called if pred() evaluates to **TRUE**, and b_2 the corresponding sequence for **FALSE**, then

$$\begin{aligned} b_1 &= \langle \text{Exec } C_1, \text{Call } S.a, \text{Exec } C_2, \text{Exec } C_p, \text{Call } S.b \rangle \\ b_2 &= \langle \text{Exec } C_1, \text{Call } S.a, \text{Exec } C_2, \text{Exec } C_p, \text{Exec } C_3 \rangle \end{aligned}$$

The job blocks of B are the set of these alternatives:

$$I_B = \{b_1, b_2\}$$

Moreover, from the **TIME** constructs one can see that $T_B = 100$ and $D_B = 20$. The complete task is modeled by the triple (I_B, T_B, D_B).

6.2.4 Notation

Some extra notation will be introduced to work with the above model.

First, because the client–server call graph is acyclic (rule 6.3), it is possible to create a strict, partial order of tasks so that A has a lower rank than B, written $A < B$, if task B makes a call to A, or makes a call to a task that makes a call to A, or so on.

Definition 6.1 (Task Rank)
Task A has a lower rank than B, denoted $A < B$, if

$$\exists a \in \Sigma_A : \text{ Call } A.a \in \Big(\bigcup I_B \cup \bigcup_{b \in \Sigma_B} I_{B.b} \Big) \tag{6.2}$$

or if

$$\exists C \in \mathbb{T}\colon A < C \wedge C < A \tag{6.3}$$

To define synchronization protocols and develop the analysis it is also necessary to describe the current run-time state of the system. If task A is currently executing the statement i, this will be written

$$A :: \; i \tag{6.4}$$

A task holds a server when its call to that server must be completed for another task to use the server. A task can hold a server either directly or indirectly, through another server:

Definition 6.2 (Holding a Server)
A task A is said to hold a server S, written $S \in \mathrm{holds}\, A$, if and only if there exists servers $S_1, S_2, \ldots, S_N$ such that

$$\begin{aligned} A &:: \mathrm{Call}\ S_1.c_1 \\ \wedge\, S_1 &:: \mathrm{Call}\ S_2.c_2 \\ \wedge\, S_2 &:: \mathrm{Call}\ S_3.c_3 \\ &\vdots \\ \wedge\, S_N &:: \mathrm{Call}\ S.c \\ \wedge\, S &:: i \end{aligned} \tag{6.5}$$

and all servers $S_1, S_2, \ldots, S_N, S$ are executing in their reply phases, handling the call of the preceding task.

Trivially, we have

$$S \in \mathrm{holds}\, A \implies S < A \tag{6.6}$$

However, it is possible that $S < A$ without ever having $S \in \mathrm{holds}\, A$, for example if $S < B < A$, but where B never calls S in the context of handling a call from A.

6.3 Synchronization Protocols

This section will define two synchronization protocol: An inheritance-based protocol and a ceiling-based protocol.

A consequence of rule 6.1 is that if a real-time task is blocked, all necessary code to unblock it will be in servers, and not in other real-time tasks.

The synchronization protocols therefore only need to change the effective scheduling priorities of server tasks.

The analysis will support both FPS and EDF. Both FPS and EDF are priority driven schedulers, so the base priority of a real-time task will be constant during each job. Servers may be given a background base priority or a "very long deadline", so that they may execute on their own when no real-time tasks are ready. The synchronization protocols will assign higher priorities to servers when they are needed by other tasks.

The minimum requirement for a synchronization protocol is that if task A is blocked waiting for a task B, then the protocol must ensure that $\pi(B) \geq \pi(A)$ (observation 3.5). Here, this condition can be expressed as

$$A :: \text{ Call } B.b \implies \pi(B) \geq \pi(A) \tag{6.7}$$

The minimum protocol which enforces this condition is an inheritance protocol, defined as follows:

Definition 6.3 (Client-server inheritance protocol)
When using the client-server inheritance protocol (CSIP), the effective priority $\pi(S)$ of a server S is given by

$$\pi(S) = \max_{X \in \mathbb{T},\, c \in \Sigma_S} \{\pi(X) \colon \;\; X :: \text{ Call } S.c\} \tag{6.8}$$

that is, the effective priority of S is the maximum priority of any task with an active or pending server call to S.

Ceiling protocols, such as the SRP, are not as straight-forward to adapt to a client–server framework. In its standard form, the SRP states that the preemption level of a resource should be set to the highest level of any task that uses that resource, whenever the resource is in use. However, in a client–server framework, a task may have to wait for a server that is not in use by another task, a situation not covered by the SRP. A solution is to use both inheritance and ceiling, with inheritance working as with the CSIP, but where each server also has a preemption level that is raised while the server executes a reply-phase:

Definition 6.4 (Client-server ceiling protocol)
The effective priority of a server when using the client-server ceiling protocol (CSCP) is given by

$$\pi(S) = \max_{X \in \mathbb{T},\, c \in \Sigma_S} \{\pi(X) \colon \;\; X :: \text{ Call } S.c\} \tag{6.9}$$

The preemption level of a server when using the CSCP is given by

$$\widehat{\pi}(S) = \begin{cases} \lceil \widehat{\pi}(S) \rceil & \text{when } \exists B \in \mathbb{T}\colon S \in \text{holds}\, B \\ \widehat{\pi}_{\min} & \text{otherwise} \end{cases} \tag{6.10}$$

where $\widehat{\pi}_{\min}$ is some preemption level lower than that of any task, and $\lceil \widehat{\pi}(S) \rceil$ is given by

$$\lceil \widehat{\pi}(S) \rceil = \max_{X \in \mathbb{T}} \{\widehat{\pi}(X)\colon S < X\} \tag{6.11}$$

As with the SRP, a task is not allowed to start a new job unless its preemption level is higher than the preemption level of all servers. For a server with its own background priority, starting to execute a request-phase without having inherited the priority of a client counts as starting a new job.

The CSCP is similar to another hybrid inheritance–ceiling protocol developed by Saksena and Karvelas [89] for UML-RT.

6.4 Schedulability Analysis

In traditional schedulability analysis, the computation of a task, C, is defined as the WCET of a job of that task. In client–server systems, part of the computation required by a real-time task will be local to servers that it uses, and not only to the real-time task itself, so the C is not as easily defined. Instead, the *demand* of a task will be defined as the worst-case execution time of one job given that it is not blocked nor preempted. This will include the WCET of a task's local code, but also the WCET of server calls made by the task.

The first thing that will be discussed in this chapter is how to actually define the WCET of a server call. This will then be used to compute the demand of a task. Preemption is then discussed. At first, preemption in client–server systems may seem to be difficult to analyze, because tasks can also be delayed indirectly by preempting servers that they are waiting for. However, it will be shown that as with traditional analysis, the worst-case delay due to being preempted is the sum of the demands of preempting tasks.

After that, worst-case blocking is computed for each of the two synchronization protocols. Finally, all of this is combined into full schedulability analyses, first for systems that use fixed priority scheduling, and then for systems that use EDF.

6.4.1 Execution Time of a Server Call

Consider some situation where

$$T :: \text{ Call } S.c \tag{6.12}$$

Both of the synchronization protocols defined in section 6.3 will ensure that $\pi(S) \geq \pi(T)$, so that no task with priority lower than $\pi(T)$ will preempt S. Ignoring preemptions from higher priority tasks for now, the execution time of the call has these three components:

1. If S has already accepted a call from another task: The time it takes for S to complete that call.

2. The time it takes for S to become ready for another server call.

3. The time it takes for S to process the server call from T.

Component 1 represents blocking in the traditional sense, because T needs to wait while the server executes code on behalf of a lower priority task.

Component 2 and 3 represent the execution time of the request and reply phase, respectively. An upper bound to component 2 is called the *worst-case request time* and is denoted Q_S, while an upper bound on component 3 is called the *worst-case reply time* and is denoted $P_{S.c}$.

In the request and reply phases T is also blocked waiting for the server. However, the server now executes code that is necessary for the completion of T. In contrast to the first component, these components need to be executed once per server call independently of whether the server is initially held by another task. It is therefore convenient to define the demand of the server call as the sum $Q_S + P_{S.c}$, and only consider the first component as blocking.

It is also clear that the execution time of component 1 will never exceed the worst-case reply time of the blocking call:

Observation 6.1. *The maximum blocking caused by a lower priority task executing* Call $S.c$ *is* $P_{S.c}$.

This is because in order to cause blocking, a server must be held by a lower priority task, which means that it must have begun the reply phase of the call from that task. If the server was not held, then it would have accepted the call of the higher priority task instead.

6.4.2 Demand

The demand of a task, server call, block or statement are all denoted by the function $\mathcal{D}(\cdot)$; the argument can be used to deduce which function is meant.

The demand of an Exec statement is the amount of computation in that statement. For a server call the demand is defined to be the sum of the worst-case request and reply times:

$$\mathcal{D}(i) = \begin{cases} r & \text{when } i = \text{Exec } r \\ Q_S + P_{S.c} & \text{when } i = \mathsf{Call}\ S.c \end{cases} \tag{6.13}$$

The demand of a block b is the sum of demands of the statements in b:

$$\mathcal{D}(b) = \begin{cases} 0 & \text{when } b = \langle\rangle \\ \mathcal{D}(i) + \mathcal{D}(b') & \text{when } b = \langle i \rangle \frown b' \end{cases} \tag{6.14}$$

The demand of a real-time task A can then be defined as the maximum demand of the job blocks of A:

$$\mathcal{D}(A) = \max_{b \in I_A} \{\mathcal{D}(b)\} \tag{6.15}$$

This definition of the demand of a task will replace the traditional notion of a task's worst-case execution time, as it is the worst-case execution time of a real-time task that is not preempted nor blocked. The worst-case request and reply times will be defined similarly:

$$Q_S = \max_{b \in I_S} \{\mathcal{D}(b)\} \tag{6.16}$$

$$P_{S.c} = \max_{b \in I_{S.c}} \{\mathcal{D}(b)\} \tag{6.17}$$

For convenience, the maximum worst-case reply time of any calls to a server will be given its own notation:

$$\widehat{P_S} = \max_{c \in \Sigma_S} P_{S.c} \tag{6.18}$$

Because servers may contain calls to sub-servers, these definitions are recursive over the set of servers. However, as long as the client–server relation graph is acyclic (rule 6.3) the recursion is guaranteed to terminate.

6.4.3 Preemption

When using mutual exclusion synchronization, a task that is preempted by a higher priority task with WCET C will be delayed by at most C. When the preempting task has completed, then the preempted task may continue from its previous state of execution.

In a client–server system the effects of preemption are more subtle. If the preempting task uses a server that the preempted task is waiting for but is not yet holding, then more computation may remain in the request phase after control returns to the preempted task, than it did at the time of the preemption. It may therefore initially seem that the delay caused by preemption may be greater than the demand of the preempting task.

However, as no computation is added to the total demand because of preemptions, the sum of demands of the preempted and preempting tasks will not change. Therefore, if a job of task L is preempted by some set of higher priority tasks $\mathcal{X}$, the total demand in the window between release to completion of L can be no greater than

$$\mathcal{D}(L) + \sum_{X \in \mathcal{X}} \mathcal{D}(X) \tag{6.19}$$

assuming that there is no blocking by tasks with lower priority than L. An example will illustrate this:

Example 6.4.1
Let L and H be real-time tasks so that $L \prec H$, and let S be a server. The tasks have the following program models:

$$\begin{aligned} I_S &= \{\langle \text{Exec } 3 \rangle\} \\ I_{S.c} &= \{\langle \text{Exec } 2 \rangle\} \\ I_L &= \{\langle \text{Exec } 2, \text{Call } S.c, \text{Exec } 2\} \\ I_H &= \{\langle \text{Exec } 2, \text{Call } S.c, \text{Exec } 1\} \end{aligned}$$

so that $\mathcal{D}(L) = 9$ and $\mathcal{D}(H) = 8$. Examples of three different scenarios in which H preempts L will be discussed:

Scenario 1: Let L be released at $t = 0$ and H at $t = 1$, as is illustrated in fig. 6.1a. In this case, L has not yet begun its server call at the time of preemption. H preempts L and executes one job. To do this, H needs to execute its local computation, which totals 3 time units, plus the server call. The server requires at most 3 time units to get ready, plus 2 for the call itself. The response time of H is therefore its demand of 8.

Task L requires 4 time units for its local computation, and like H, $3+2$ units are required for the server call itself. The delay caused by the interference from H is 8. Adding up, the response time of L becomes 17.

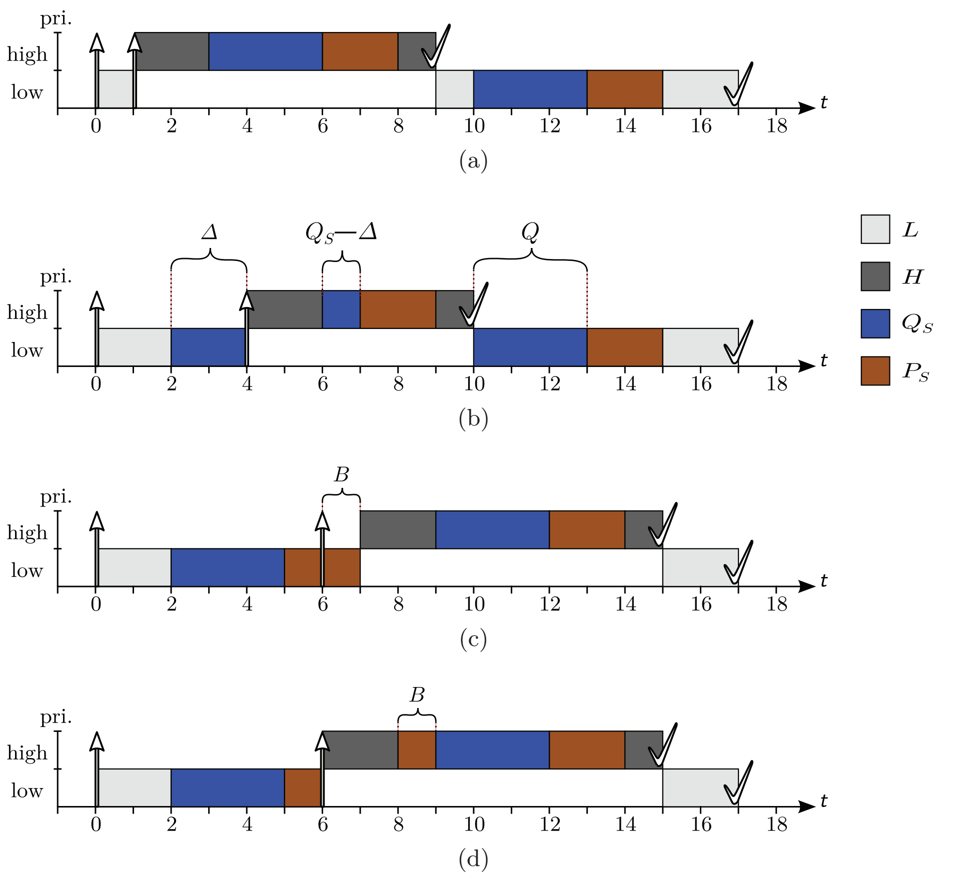


Figure 6.1: Example 6.4.1: Task preemption and server access. H preempts L when
(a) server is at beginning of request phase
(b) server is in the middle of request phase
(c) server is in the middle of reply phase (CSCP)
(d) server is in the middle of reply phase (CSIP).

Scenario 2: Say instead that S was in the middle of its request phase when H preempted L, and that Δ out of Q_S had already been executed (fig. 6.1b). The remaining request phase execution time for H is then at most $Q_S - \Delta$. H saves Δ of execution time compared to case 1, and its response time decreases accordingly.

When H completes, S is left at the beginning of the request phase. L therefore has to wait for a full new request phase before the call is accepted, adding Δ more execution time to the server call compared to case 1. Nevertheless, it suffers correspondingly less delay from interference, and its response time remains the same.

Scenario 3: Now consider the case where H is released while S is in its reply phase, serving L. Say that S has B units of computation remaining in its reply phase. If using the CSCP, H will not be able to start until S has completed its reply phase, the start of H being delayed B time units due to blocking (fig. 6.1c). If using the CSIP, this delay instead occurs when H attempts to call S (fig. 6.1d). In either case, the server will first complete the server call from L with the priority of H before H is allowed to progress, causing H to be blocked for B time units, increasing its response time.

The job of L completes at the same time in all scenarios. This is because the total computational demand from the release to completion of L is unchanged: It is the sum of the computation local to L and H, plus one request and reply phase for each call to S.

6.4.4 Blocking under the Ceiling Protocol

As discussed in section 6.4.1, the blocking caused by a lower priority task holding a server is at most equal to the server's worst-case reply time. To perform schedulability analysis one must also know which server, or servers, that cause the maximum blocking when held.

These worst-case blocking terms will now be developed, for both the CSCP and the CSIP. The blocking terms will be defined with preemption levels so that they can be used directly with both EDF and FPS based analysis.

Definition 6.5 (Blocking term B_π)
Let $\mathcal{W}$ be any window in time in which there is always a real-time task with preemption level higher than or equal to π ready to execute. The blocking term B_π is defined as an upper bound to the amount of execution time in

$\mathcal{W}$ performed by servers held by tasks with a preemption level lower than π.

With the CSCP, only one lower level task can block a continuous set of higher level tasks:

Lemma 6.1 (CSCP). *The blocking term when using the* CSCP *is bounded by*

$$B_\pi \leq \max \widehat{P_S} \tag{6.20}$$

where $S \in \mathbb{S}$ *satisfies*

$$\widehat{\pi}(A) < \pi \leq \lceil \widehat{\pi}(S) \rceil \quad \wedge \quad S < A \tag{6.21}$$

for some $A \in \mathbb{T}$*.*

Proof. The blocking term is defined as the amount of execution time required by servers held by tasks with lower levels than π. When using the CSCP, if a server S held by some lower level task A executes in $\mathcal{W}$ it can only be because its level has been raised to π or higher; necessary conditions for this to happen are given by eq. (6.21).

After A acquired S, no task C with $\widehat{\pi}(C) \leq \lceil \widehat{\pi}(S) \rceil$ would be allowed to preempt it or any servers that it might use. By transitivity this includes all tasks C' with $\widehat{\pi}(C') \leq \pi$. Therefore, at any point in time there can only be one task A that satisfies eq. (6.21). S is therefore never blocked, and will execute for at most $\widehat{P_S}$ time before its ceiling is dropped. This yields

$$B_\pi \leq \widehat{P_S} \tag{6.22}$$

After A has finished its server call, no lower level tasks hold any servers used by tasks that execute in the window, so after this, no higher level tasks will be blocked. The highest value for B_π is found by maximizing eq. (6.22) while satisfying eq. (6.21). This results in lemma 6.1. □

6.4.5 Blocking under the Inheritance Protocol

With the CSIP it is possible for several low level tasks to hold servers that block higher level tasks. However, in this section it will be shown that because all server accesses are strictly nested, a safe assumption is for each task to hold only one server, as long as this server is chosen to be the one with the greatest worst-case reply time. The WCET required to free this server must then include the time needed to free other servers that the same low level task may hold.

Lemma 6.2. *Let $\mathcal{W}$ be a window in time in which there is always a real-time task with preemption level higher than or equal to π ready to execute. Let X be a task with $\widehat{\pi}(X) < \pi$ and let $B_{\pi,X}$ be the maximum amount of work executed by servers held by X in $\mathcal{W}$. Then*

$$B_{\pi,X} \leq \max_{S \in \mathcal{S}} \widehat{P}_S \tag{6.23}$$

where $\mathcal{S}$ is the set of servers held by X that are also used by any of the higher level tasks.

Proof. Let the state of X and the servers held by X be

$$\begin{aligned} X &:: \text{Call } S_1.c_1 \\ \wedge\, S_1 &:: \text{Call } S_2.c_2 \\ \wedge\, S_2 &:: \text{Call } S_3.c_3 \\ &\vdots \\ \wedge\, S_N &:: i \end{aligned} \tag{6.24}$$

so that $\mathsf{holds}\, X = \{S_1, S_2, \ldots, S_N\}$, and S_N does not hold any servers. Let b_i denote the remaining block of computation required to complete the reply phase of S_i that is local to S_i.

The only reason why a server held by X is executed in the window is because a higher priority task also requires it. Higher priority tasks may call servers held by X in any order. Lemma 6.2 only describes execution time by servers held by X; so transitive blocking by further lower priority tasks is by definition not included.

Let S_h be the highest ranking server in $\mathcal{S}$. Completing the call to S_h directly when there is no transitive blocking requires $\sum_{k=h}^{N} \mathcal{D}(b_k)$ of execution. This must by definition by less than the worst-case reply time of S_h, which by definition must be less than the greatest worst-case reply time of any server in $\mathcal{S}$:

$$\sum_{k=h}^{N} \mathcal{D}(b_k) \leq P_{S_h.c_h} \leq \max_{S \in \mathcal{S}} \widehat{P}_S \tag{6.25}$$

A stronger version of the lemma will now be proved, namely that

$$B_{\pi,X} \leq \sum_{k=h}^{N} \mathcal{D}(b_k) \tag{6.26}$$

Proof by induction. As a basis, say S_a is the first server call that needs to be completed. This requires a worst-case execution time of

$$\sum_{k=a}^{N} \mathcal{D}(b_k) \tag{6.27}$$

by servers held by X. Equation (6.26) therefore holds for a first server call by a higher level task. After this, a is the highest ranking task that a higher level task has required, and X now holds the servers $\{S_1, \dots, S_{a-1}\}$.

For the inductive step, say after some such calls that S_a is the highest ranking server for which the call has already been completed. X then holds servers $\{S_1, \dots, S_{a-1}\}$, and, assuming eq. (6.26), completing the previous calls have taken a maximum time of $\sum_{k=a}^{N} \mathcal{D}(b_k)$.

Say the next server to be completed is S_b. If S_b has a lower rank, ie, $S_b < S_a$, which implies $b > a$, then S_b must already have been completed in previous calls. If S_b has a higher rank, ie, $b < a$, then it takes

$$\sum_{k=b}^{a-1} \mathcal{D}(b_k) \tag{6.28}$$

to complete, because servers from S_a to S_N have already completed their reply phases. In total, completing this call to S_b plus all previous server calls take at most

$$\sum_{k=b}^{a-1} \mathcal{D}(b_k) + \sum_{k=a}^{N} \mathcal{D}(b_k) = \sum_{k=b}^{N} \mathcal{D}(b_k) \tag{6.29}$$

so eq. (6.26) holds. As eq. (6.26) holds for the first server call, and, given that it holds for a series of server calls it also holds for the next; by induction the equation must hold for all sequences of server calls from higher priority tasks.

This proves eq. (6.26). It can be seen from eq. (6.25) that this is a stronger version of the lemma, so the lemma must also hold. □

Lemma 6.3 (CSIP). *The blocking term when using the* CSIP *is bounded by*

$$B_\pi \leq \max_{f\colon \mathbb{T}' \rightarrowtail \mathbb{S}} \sum_{X \in \operatorname{dom} f} C_\pi(X, f(X)) \tag{6.30}$$

where $f\colon \mathbb{T}' \rightarrowtail \mathbb{S}$ is a partial, injective function[1] *from tasks to servers,* dom f

[1] *Partial* implies that f may not be defined for all tasks. *Injective* implies that no two tasks are mapped to the same server.

is the domain of f and $C_\pi(X,S)$ is given by

$$C_\pi(X,S) = \begin{cases} \widehat{P}_S & \textit{if} \quad \exists T : \widehat{\pi}(X) < \pi \leq \widehat{\pi}(T) \land S < T \land S < X \\ 0 & \textit{otherwise} \end{cases} \tag{6.31}$$

Informally, $C_\pi(X,S)$ is the maximum time server S may block a task with level higher than π because it is held by the lower level task X.

Proof. Consider the blocking term B_π as defined in definition 6.5. If a server held by a task with lower level than π executes in the window, it must be because the server is required by a task with a level higher than or equal to π. It may be called directly by such a task, or it may be required indirectly, in order to complete an ongoing call a server required by the task. For a server S held by a lower level task X to execute in the window, the following points must therefore hold:

1. X must be lower level: $\widehat{\pi}(X) < \pi$.
2. S must be held by X, which implies $S < X$.
3. S must be used by some task T with a level higher than or equal to π, or by a server used by T, both which implies $S < T$ for some $T : \widehat{\pi}(T) \geq \pi$.

Only if all of the above points hold may S be a server held by a lower level task X that will execute in $\mathcal{W}$. According to lemma 6.2, if X satisfies the first criterion, and $\mathcal{S}$ is a set of servers that satisfy the second and third criteria, then these servers will execute at most $\max_{S \in \mathcal{S}} \widehat{P}_S$ in $\mathcal{W}$.

Say each lower level task X is assigned an arbitrary, but mutually exclusive set of servers $\mathcal{S}_X$. Let $C'_\pi(X,\mathcal{S}_X)$ denote the execution in $\mathcal{W}$ of servers in $\mathcal{S}_X$ held by X, only if X is a lower level task, and only for servers that it is possible that X may hold. Then

$$C'_\pi(X,\mathcal{S}_X) \leq \max_{S \in \mathcal{S}_X} \begin{cases} \widehat{P}_S & \text{if} \quad \exists T : \widehat{\pi}(X) < \pi \leq \widehat{\pi}(T) \land S < T \land S < X \\ 0 & \text{otherwise} \end{cases} \tag{6.32}$$

this can be rewritten as

$$C'_\pi(X,\mathcal{S}_X) \leq \max_{S \in \mathcal{S}_X} C_\pi(X,S) \tag{6.33}$$

where

$$C_\pi(X,S) \leq \begin{cases} \widehat{P}_S & \text{if} \quad \exists T : \widehat{\pi}(X) < \pi \leq \widehat{\pi}(T) \land S < T \land S < X \\ 0 & \text{otherwise} \end{cases} \tag{6.34}$$

Worst-case blocking is achieved when each lower level task X is assigned a set $\mathcal{S}_X$ that maximizes the sum of execution of servers held by lower level tasks in $\mathcal{W}$. If this maximum is denoted B_π, then

$$B_\pi \le \max_{g\colon \mathbb{T} \rightarrowtail \{\mathbb{S}\}} \sum_{X \in \mathbb{T}} \max_{S \in g(X)} C_\pi(X, S) \tag{6.35}$$

where g is a function from tasks to sets of servers so that

$$\forall A, B \in \mathbb{T}\colon A \neq B \implies g(A) \cap g(B) = \emptyset \tag{6.36}$$

Because only one server from each set contributes to the blocking value, the maximum value can always be achieved by assigning only one server to each task. This yields

$$B_\pi \le \max_{f\colon \mathbb{T}' \rightarrowtail \mathbb{S}} \sum_{X \in \operatorname{dom} f} C_\pi(X, f(X)) \tag{6.37}$$

where $f\colon \mathbb{T}' \rightarrowtail \mathbb{S}$ means that

$$\forall A, B \in \mathbb{T}\colon A \neq B \implies f(A) \neq f(B) \tag{6.38}$$

and that f is not necessarily defined for all tasks. This proves the lemma. □

Immediately it may seem to be difficult to actually find the maximal assignment function f so that the worst-case blocking can be found. Fortunately, eq. (6.37) has the form of a *linear sum assignment problem*, which is solvable by for example the Hungarian Algorithm. The worst-case complexity is $O(|\mathbb{T}|^3)$. For details and algorithms see for example Burkard and Çela [25].

6.4.6 Fixed Priority Scheduling and Response-Time Analysis

The above blocking term computations can be used to create a RTA-based schedulability test for synchronous client–server systems. Within a task's worst-case response time R, it must complete its own execution and wait for a sufficient number of instances of higher priority tasks. It must also wait for blocking lower priority tasks to complete calls to shared servers:

Theorem 6.4. *In a synchronous client–server system with a fixed priority scheduler, the worst-case response-time of a real-time task A satisfies*

$$R_A = B_{\widehat{\pi}(A)} + \mathcal{D}(A) + \sum_{X \in \mathbb{T}\colon P_X > P_A} \left\lceil \frac{R_A}{T_X} \right\rceil \mathcal{D}(X) \tag{6.39}$$

where $B_{\widehat{\pi}(A)}$ is the worst-case blocking term given by either lemma 6.1 or lemma 6.3, depending on the synchronization protocol.

Proof. During the window between the release of a job of A, and its latest completion, R_A, the only tasks that will get to execute are A and higher priority tasks, and servers required by A or by higher priority tasks.

Equation (6.39) is the sum of the demands of these tasks, divided into three terms:

1. The first term is blocking. The time interval from the release to worst-case response time of A is a window in time $\mathcal{W}$ where there is always a task with preemption level equal to or higher than $\widehat{\pi}(A)$ ready to execute, so lemmas 6.1 and 6.3 applies to $\mathcal{W}$. $B_{\widehat{\pi}(A)}$ is then the worst-case execution time by servers held by lower priority tasks in $\mathcal{W}$. Note that priorities are equal to preemption levels under FPS.

2. A itself requires at most $\mathcal{D}(A)$ of demand.

3. Each higher priority task $X\colon P_X > P_A$ has at most $\left\lceil \frac{R_A}{T_X} \right\rceil$ jobs that overlap with the execution of A. Each such job has a demand of $\mathcal{D}(X)$. The sum of all these demands for all higher priority tasks yield the last term in eq. (6.39).

No other task will execute until A has completed, and will therefore not affect the response-time of A. □

Theorem 6.4 have the same form as Theorem 5.5, which is used for RTA of systems that use mutual exclusion synchronization. When having computed blocking terms and demands, the same methods that can be used for solving the standard RTA equation, can also be used for eq. (6.39).

6.4.7 Earliest Deadline First

The schedulability test for synchronous client–server systems scheduled under EDF is based on the PDC.

Theorem 6.5. *A synchronous client–server system scheduled with* EDF *is schedulable if*

$$\forall l \in \mathbb{R}^+ : \text{DBF}(l) + B_{\widehat{\pi}(l)} \le l \tag{6.40}$$

where $B_{\widehat{\pi}(l)}$ is the blocking term for the preemption level equivalent to a relative deadline of l, given by either lemma 6.1 or lemma 6.3 depending on the synchronization protocol, and DBF(l) *is the demand-bound function:*

$$\text{DBF}(t) = \sum_{X \in \mathbb{T}} \left(1 + \left\lfloor \frac{t - D_X}{T_X} \right\rfloor\right) \mathcal{D}(X) \tag{6.41}$$

Proof. Proof by contradiction. Assume that eq. (6.40) holds, but that there is a first deadline miss at time d_m by job x. Let $\mathcal{W}$ be the window $[t_0, d_m]$, where t_0 is chosen as early as possible, but so that there is always a task with deadline earlier than d_m ready to execute in $\mathcal{W}$.

The tasks that get to execute within $\mathcal{W}$ are therefore only the real-time tasks with deadline within $\mathcal{W}$ or servers required by those tasks. The worst-case cumulative demand by these real-time tasks, ignoring blocking, is no greater than $\textsc{dbf}(l)$ by definition of the demand function.

A task that causes blocking must have started its execution before t_0 in order to hold a server. Also, its absolute deadline must be after the window, otherwise its execution should have been part of $\mathcal{W}$. Consequently, its relative deadline must be greater than l. Under EDF, preemption levels are ordered by relative deadlines, so $\mathcal{W}$ satisfies the requirement for lemmas 6.1 and 6.3 of a window where there is always a task with preemption level higher than or equal to $\widehat{\pi}(l)$ ready to execute. Therefore, the term $B_{\widehat{\pi}(l)}$ is an upper bound to the time servers held by tasks with deadline outside the window will execute during the window.

The total required demand during $\mathcal{W}$ for jobs that have deadlines earlier than or equal to d_x is therefore no greater than

$$\textsc{dbf}(l) + B_{\widehat{\pi}(l)} \tag{6.42}$$

This value must be greater than the window length, otherwise x would not have missed its deadline, as was the initial premise. If eq. (6.40) holds, then no such window can exist and the system cannot be unschedulable. □

6.5 Application Examples

Two examples of using the analysis will be demonstrated. First, it will be shown how the inheritance blocking term in lemma 6.3 can be applied to protected objects, and that it yields a lower worst-case blocking than existing methods for inheritance. Second, it will be shown how computation in server calls can be deferred, a transformation that may sometimes increase schedulability, and which is difficult to accomplish without using synchronous communication.

6.5.1 Applying the Inheritance Blocking Term to Protected Objects

Protected objects can always be implemented using synchronous client–server communication [73]. Here, a protected object will be modeled as a

Table 6.1: Improved Inheritance Blocking Term. Table shows worst case blocking of high priority task caused by lower priority task L holding resource r.

	Resource r					
Task X	r_1	r_2	r_3	r_4	r_5	r_6
A	9	8	10	0	6	5
B	9	0	10	7	0	0
C	0	3	0	0	0	3
D	1	1	1	1	1	1

server with a worst-case reply-time equal to the critical section length of the protected object, and a worst-case request time of 0.

Existing methods for computing the blocking term in systems using the PIP do this by summing up the worst-case critical section length of all resources (eq. (5.14)). The blocking term presented in this paper is more complex, and instead tries to find the worst-case assignment of lower priority tasks to resources (lemma 6.3). This sometimes yields a lower bound on worst-case blocking. An example will be given:

Example 6.5.1
A system consists of tasks A, B, C, D and H, and six protected-object type resources $r_1, r_2, \ldots, r_6$. The system is scheduled using FPS, with H as the highest priority task. The maximum blocking of H due to sharing resource r with each lower priority task is given in table 6.1. A zero indicates that the resource is not shared with the corresponding task.

Using the traditional method for computing worst-case blocking, which sums up the worst-case critical section length of all resources, results in a blocking term of $B_H = 45$. However, this erroneously assumes that all resources can be held at the same time with their worst-case critical sections. When critical sections are strictly nested, which is the case with protected objects, this is too pessimistic.

An improved B_H is found with lemma 6.3. The maximization problem is equivalent to selecting one cell from each row in table 6.1 without using the same column twice, while maximizing the sum of these cells. Using an appropriate algorithm one gets the following maximal assignment: $r_1 \leftarrow B$, $r_2 \leftarrow D$, $r_3 \leftarrow A$, $r_6 \leftarrow C$, which results in A blocking of $B_H = 23$. Note that the maximum blocking with a ceiling protocol would be even lower, with $B_H = 10$.

Listing 6.3: *Data Structure Server with Immediate Write*

```
loop
   select
      accept Read (Key : in Key_Type; An_Item : out Data_Type) do
         -- Read item from data structure
      end Read;
   or
      accept Write (Key : in Key_Type; An_Item : in Data_Type) do
         -- Write item to data structure
      end Write;
   end select;
end loop;
```

6.5.2 Deferred Server Calls

One benefit of using synchronous, client–server based communication is that in some cases, blocking can be reduced by deferring computation in a server call to the request phase of the succeeding call. Part of a server call can be deferred whenever its result is not needed for computing the return value of the call.

The effect of deferring computation in a call to server S is that work required by that call is moved from the reply-phase to the request-phase preceding the next call. This will decrease the reply-time of that call, which may or may not decrease the worst-case reply time of the server. Moreover, it may increase the server's worst-case request-time, though not necessarily by the same amount. For a real-time task A that uses S, the part of the demand $\mathcal{D}(A)$ that is derived from a call to S is a function of $Q_S + P_{S.c}$ for each call c made to S. However, blocking by A of higher level tasks is a function of $\widehat{P_S}$ only, and thus a decrease in $\widehat{P_S}$ at the cost of an increase in Q_S may decrease the blocking experienced by higher priority tasks that are also using S, making them easier to schedule. The effect is illustrated in the following example:

Example 6.5.2
A fixed-priority system consists of two implicit deadline real-time tasks, A

Listing 6.4: *Data Structure Server with Deferred Write*

```
loop
    select
        accept Read (Key : in Key_Type; Data : out Data_Type) do
            -- Read item from data structure
        end Read;
    or
        accept Write (Key : in Key_Type; Data : in Data_Type) do
            To_Be_Written_Key = Key;
            To_Be_Written_Data = Data;
        end Write;
    end select;
    if Non_Null(To_Be_Written_Key) then
        -- Write item to data structure
    end if;
end loop;
```

and B, with the following properties:

$$\begin{aligned} P_A &> P_B \\ T_A = D_A &= 20 \\ T_B = D_B &= 50 \\ I_A = I_B &= \{\langle \text{Exec } 1, \text{ Call } S.r, \text{ Exec } 1\rangle, \\ &\qquad \langle \text{Exec } 1, \text{ Call } S.w, \text{ Exec } 1\rangle\} \end{aligned} \tag{6.43}$$

Tasks A and B are identical, except for their periods, and both use a server S that holds a shared data structure. S has two operations, a quick read operation ($S.r$) with a WCET of 1, and a slow write operation ($S.w$) with a WCET of 10. As expressed by eq. (6.43), each job of A and B uses either one of these operations, but never both in the same instance.

Let the server be implemented so that both the read and write operations are part of the calls (ie, non-deferred) as illustrated in listing 6.3. An idealized model of this server is

$$\begin{aligned} I_S &= \{\langle\rangle\} & Q_S &= 0 \\ I_{S.r} &= \{\langle \text{Exec } 1\rangle\} & P_{S.r} &= 1 \\ I_{S.w} &= \{\langle \text{Exec } 10\rangle\} & P_{S.w} &= 10 \end{aligned}$$

Using theorem 6.4, the response time of A can be computed as

$$\begin{aligned} \mathcal{D}(A) &= 12 \\ B_{\widehat{\pi}(A)} &= P_{S.w} = 10 \\ R_A &= \mathcal{D}(A) + B_{\widehat{\pi}(A)} = 22 \end{aligned}$$

This is clearly infeasible as $D_A = 20$.

Notice that task A may have to execute the heavy $S.w$ operation twice in a single instance, one for itself and one if it preempts B's own write operation at an early stage. It is possible to relieve A of this extra write operation, by deferring it and executing the bulk of it after the entry call. An implementation with a deferred write is illustrated in listing 6.4. Now:

$$\begin{aligned} I_S &= \{\langle\rangle, \langle \text{Exec } 10\rangle\} & Q_S &= 10 \\ I_{S.r} &= \{\langle \text{Exec } 1\rangle\} & P_{S.r} &= 1 \\ I_{S.w} &= \{\langle\rangle\} & P_{S.w} &= 0 \end{aligned}$$

Notice how the sum $Q_S + \widehat{P_S}$ has increased, but that the worst-case reply-time has decreased. The response time of A can now be computed again:

$$\begin{aligned} \mathcal{D}(A) &= 13 \\ B_{\widehat{\pi}(A)} &= P_{S.r} = 1 \\ R_A &= \mathcal{D}(A) + B_{\widehat{\pi}(A)} = 14 \end{aligned}$$

This is less than D_A, so A will not miss deadlines. The response time of B is given by

$$\begin{aligned} &\mathcal{D}(B) = 13 \\ &R_A = \mathcal{D}(B) + \left\lceil \frac{R_B}{T_A} \right\rceil \mathcal{D}(A) = 13 + \left\lceil \frac{R_B}{20} \right\rceil 13 \end{aligned}$$

which has a least fixed point in $R_B = 39$. As both A and B have acceptable response-times, the system is now schedulable. A similar effect would be much more difficult achieve if tasks communicate using protected objects or other type of shared-memory based synchronization, because computation cannot be moved out of a critical section while maintaining mutual exclusion.

6.6 Discussion

To allow for analysis of synchronously communicating systems, it must be ensured that no tasks will require synchronization with tasks that have not yet been released. In Burns and Wellings [26] this was accomplished by restricting synchronous communication to task instances with simultaneous release. In Saksena and Karvelas [89] it was done by decoupling real-time tasks (the external events) from the tasks performing the synchronization (the capsules). In this chapter, the same was achieved by requiring tasks to communicate in a client–server fashion, while prohibiting real-time tasks from acting as servers.

To be amendable to the analysis of this chapter, the client–server communications must satisfy several rules. In addition to the rule preventing real-time tasks from acting as servers, other rules help guarantee the absence of deadlocks.

Rule 6.1, which prohibits any computation on the client-side between the request to a server and the corresponding reply, was added merely to simplify the analysis. To guarantee the absence of deadlocks it would be sufficient to deny communication on the client-side. The justification for this rule is that it is implicitly followed by Ada programs, where no such computation can be expressed. It is therefore only a real restriction when using formalisms that allow more complex communication patterns.

The analysis allows schedulability to be analyzed for systems that use synchronous communication, without being so restrictive as to remove the advantages of synchronous communications such as modular composition and safety. Servers are allowed to execute code when not serving a client, and are thus much more powerful than a mere synchronous implementation of a protected object. For the situations where protected objects are required they can still be used, and modeled during the schedulability analysis as a server with no request-phase. Moreover, when using protected objects and the PIP, the analysis in this chapter is less pessimistic than existing methods and is therefore an improvement to these methods even for systems that only use protected objects for communication.

It was shown that the schedulability of a system may sometimes be improved by deferring computation generated by a server call until after the call, an optimization which is difficult to achieve when using mutual exclusion based communication. This is because the client–server framework separates the notion of mutual exclusion—data local to a server—from the notion of a critical section—code that causes blocking, in this case the reply phase. For some systems this transformation improves schedulability, but for others it may reduce it. It is not currently known when the transformation helps, and when it does not, and this would be an interesting topic for further study.

The analysis only applies to uniprocessor systems. One reason for this is that existing multiprocessor synchronization protocols have a basic premise of short critical sections, an assumption which does not hold in a client–server based system. Another reason is that multiprocessor analysis typically prohibits JLP, and thus limits the parallelism of process-oriented systems. This is not an issue in uniprocessor systems because each process is serialized during execution, so that enabling JLP does not affect schedulability. No complete analysis of synchronously communicating multiprocessor systems is given in this thesis, but the next chapters will begin this work by developing support for JLP.

Chapter 7

Modeling Job-Level Parallelism

He gave her cat food

ANON

A NOTABLE feature of process-oriented systems is the use of parallelism as a means for program organization. To accommodate this, jobs in process-oriented real-time systems should be allowed to have a free parallel structure, that is, an arbitrary combination of sequential and parallel parts. Moreover, to increase efficiency on multiprocessor platforms, each job should also be allowed to execute on multiple processors *simultaneously*. This, however, is not widely supported by existing analyses.

This chapter introduces an abstract process model for describing the computational requirements of real-time jobs with such a parallel structure. The model abstracts away any notion of what a process actually does, leaving only its parallel structure and the computation time required to complete each part of that structure. This kind of abstract process will be called a computation time process (CTP).

7.1 Introduction

The CTPs may be used to analyze real-time systems with job-level parallelism (JLP) and malleable jobs; ie, systems where each real-time job is allowed to execute simultaneously on multiple processors, and where the number of processors assigned to each job may change during the execution of the job.

The model considers just the **SEQ/PAR** structure for parallelism: processes are explicitly specified to execute in parallel or in sequence, and a parallel process does not terminate until all the sub-processes terminate.

The sub-processes themselves may also have a **SEQ/PAR**-structure. This is the structure of parallelism used in occam and its derivatives.

Although simple channel communications can be modeled by using sequence restrictions to form synchronization points, CTPs cannot be used to model communication in general, and it is therefore not possible to derive a CTP from a general program written in for example Toc. CTPs should instead be considered building blocks that can be used as a basis for a complete timing analysis.

The model will assume discrete time, and a work-conserving intra-job scheduler, ie, an intra-job scheduler that does not keep processors idle while there is computation that may be executed.

Here, each CTP is analyzed in isolation. Still, it will always be assumed that the process is executing in some environment where processors are shared with other tasks. Some of these tasks will affect the number of processors available to the process under consideration, so the number of available processors will be considered time varying and non-deterministic. To allow schedulability analysis of a complete system one would need to analyze how different CTPs interact when executed on the same platform. The results from this chapter are necessary as a first step towards such an analysis.

7.1.1 Outline

The structure of this chapter is as follows: The CTP model is defined in section 7.2. Two partial orders over CTPs are defined in section 7.3. These orders give a precise definition to what it means for a process to be an upper bound of another with respect to worst-case computation time, and what it means for a process to be easier to schedule than another.

In section 7.4 it is demonstrated that some CTPs exhibit counterintuitive behavior, and that this behavior may occur in all systems where an arbitrary **SEQ/PAR**-structure is allowed. For instance, even if schedulability analysis of a system finds that all tasks meet their deadlines, this may no longer be true if a task executes less than its expected amount of computation. In section 7.5 a subset of well-behaved CTPs is identified that never exhibit this kind of behavior.

7.2 Computation Time Processes

This section will define the CTP model.

7.2.1 Definitions

The set of CTPs will be denoted $\mathbb{P}$. A process $P \in \mathbb{P}$ may be one of the primitive processes $\mathbf{0}$ or $\mathbf{1}$, or a combination of two other processes using either the sequence operator "$;$", or the parallel operator "$\|$":

$$\begin{aligned} P \in \mathbb{P} \iff & P = \mathbf{1} & & \\ & \vee P = \mathbf{0} & & \\ & \vee P = Q\,;\,R & & Q, R \in \mathbb{P} \\ & \vee P = Q \,\|\, R & & Q, R \in \mathbb{P} \end{aligned} \tag{7.1}$$

The zero process $\mathbf{0}$ denotes a process which requires no work. It will never consume CPU, and any process following in sequence after $\mathbf{0}$ may start immediately.

The unit process $\mathbf{1}$ denotes a process which requires one unit of computation. The value of one unit of computation with respect to real time represents the minimum quantification of time for the system.

The sequence process $P\,;\,Q$, where $P, Q \in \mathbb{P}$, denotes a process of two subprocesses where one has to terminate execution before the other can begin. It satisfies the following basic laws:

$$\mathbf{0}\,;\,P = P \tag{left-identity}$$
$$P\,;\,\mathbf{0} = P \tag{right-identity}$$
$$(P\,;\,Q)\,;\,R = P\,;(Q\,;\,R) \tag{associativity}$$

The parallel process $P \,\|\, Q$, where $P, Q \in \mathbb{P}$, represents two processes that may execute concurrently. It satisfies the following basic laws:

$$\mathbf{0} \,\|\, P = P \tag{identity element}$$
$$P \,\|\, Q = Q \,\|\, P \tag{commutativity}$$
$$(P \,\|\, Q) \,\|\, R = P \,\|(Q \,\|\, R) \tag{associativity}$$

7.2.2 Measures on Computation Time Processes

An important property of any computation time process is its total amount of computation. This will be described as a function $C\colon \mathbb{P} \to \mathbb{N}$, defined by

$$\begin{aligned} C(\mathbf{1}) &= 1 \\ C(\mathbf{0}) &= 0 \\ C(P\,;\,Q) &= C(P) + C(Q) \\ C(P \,\|\, Q) &= C(P) + C(Q) \end{aligned} \tag{7.2}$$

For a process P, if the number of processors available to P is always greater or equal to the number of processors P can use, then it will be the longest sequence in P that determines its execution time. The length of a process, $\mathcal{L}\colon \mathbb{P} \to \mathbb{N}$, describes this time:

$$\begin{aligned}
\mathcal{L}(\mathbf{1}) &= 1 \\
\mathcal{L}(\mathbf{0}) &= 0 \\
\mathcal{L}(P\,;\,Q) &= \mathcal{L}(P) + \mathcal{L}(Q) \\
\mathcal{L}(P \,\|\, Q) &= \max\{\mathcal{L}(P), \mathcal{L}(Q)\}
\end{aligned} \tag{7.3}$$

The length of a process is thus the minimum execution time of the process.

Note that neither of the above definitions (of C and $\mathcal{L}$) includes time for computational overheads in managing the parallel processes. These could be accounted for by adding a sequence of unit processes before and after the parallel composition, but are not addressed here for simplicity.

Another important property is the immediate height of the process, $\mathcal{H}\colon \mathbb{P} \to \mathbb{N}$, defined as the number of parallel branches that are ready to be executed:

$$\begin{aligned}
\mathcal{H}(\mathbf{1}) &= 1 \\
\mathcal{H}(\mathbf{0}) &= 0 \\
\mathcal{H}(P\,;\,Q) &= \begin{cases} \mathcal{H}(P) & \text{if } P \neq \mathbf{0} \\ \mathcal{H}(Q) & \text{if } P = \mathbf{0} \end{cases} \\
\mathcal{H}(P \,\|\, Q) &= \mathcal{H}(P) + \mathcal{H}(Q)
\end{aligned} \tag{7.4}$$

The height of a process is therefore the maximum number of processors the process can utilize for the first step of its computation. This information is needed when scheduling a process to the number of processors available for that first step.

7.2.3 Computing a Single Step

Time is modeled as discrete, so processes are executed in unit time steps. After each execution step, the resulting process is a function of the original process and the number of processors assigned to it for that step. If a process consists of multiple parallel branches and there are too few processors available to execute them all, then the scheduler must choose a subset of branches to execute. Thus, if nothing is known about the details of the scheduler itself, the result of an execution step may be considered non-deterministic. The only detail of the scheduler that will be assumed is that it is work-conserving, meaning that it does not keep any processors idle if there is ready work to be done.

The function step: $\mathbb{P} \times \mathbb{N} \to \{\mathbb{P}\}$ takes a process P and a number of processors m and returns the set of all possible processes that can result from executing a single step of P with m processors. An explicit formulation of the step function can be written as

$$
\begin{aligned}
\mathrm{step}(\mathbf{1}, m) &= \begin{cases} \{\mathbf{1}\} & \text{if } m = 0 \\ \{\mathbf{0}\} & \text{if } m \geq 1 \end{cases} \\
\mathrm{step}(\mathbf{0}, m) &= \{\mathbf{0}\} \\
\mathrm{step}((P\,;\,Q), m) &= \begin{cases} \{(P'\,;\,Q) \colon P' \in \mathrm{step}(P, m)\} & \text{if } P \neq \mathbf{0} \\ \mathrm{step}(Q, m) & \text{if } P = \mathbf{0} \end{cases} \\
\mathrm{step}((P \,\|\, Q), m) &= \Big\{(P' \,\|\, Q') \colon P' \in \mathrm{step}(P, m_P), Q' \in \mathrm{step}(Q, m_Q), \\
&\qquad m_P \in [0, \mathcal{H}(P)], \\
&\qquad m_Q \in [0, \mathcal{H}(Q)], \\
&\qquad m_P + m_Q = \min\{\mathcal{H}(P) + \mathcal{H}(Q), m\}\Big\}
\end{aligned}
\tag{7.5}
$$

where the last equation states that a scheduler can distribute the available processors to the parallel branches P and Q in several ways, as long as the amount of execution performed by the step is the maximum possible when limited by the number of parallel branches and by the number of available processors.

If the resulting set from a step of a process has only one member, then the execution of the process was deterministic; if the resulting set has more than one member, then the result of execution depends on choices made by the scheduler. Two examples of using the step-function are given below:

Example 7.2.1
Let a process P be given by

$$P = \mathbf{1} \,\|\, \mathbf{1}$$

In this case, $\mathcal{H}(P) = 2$, because for its first step, P may utilize two processors.

Say 3 processors are available for P for its first step, ie, $m = 3$. Then,

$$\begin{aligned}
\text{step}((\mathbf{1} \,\|\, \mathbf{1}), 3) &= \Big\{(P' \,\|\, Q') \colon P' \in \text{step}(\mathbf{1}, m_P), Q' \in \text{step}(\mathbf{1}, m_Q), \\
&\qquad m_P \in [0,1], m_Q \in [0,1], m_P + m_Q = \min\{2,3\}\Big\} \\
&= \Big\{(P' \,\|\, Q') \colon P' \in \text{step}(\mathbf{1}, 1), Q' \in \text{step}(\mathbf{1}, 1)\Big\} \\
&= \Big\{(P' \,\|\, Q') \colon P' \in \{\mathbf{0}\}, Q' \in \{\mathbf{0}\}\Big\} \\
&= \{\mathbf{0} \,\|\, \mathbf{0}\} \\
&= \{\mathbf{0}\}
\end{aligned}$$

which means that P will complete all its computation after one time step if 3 processors are available for it to use (P would also have completed if given 2 processors).

Example 7.2.2
Let a process P be given by

$$P = \mathbf{1} \,\|\, (\mathbf{1} \,;\, \mathbf{1})$$

Say P is given one processor, ie, $m = 1$. Then,

$$\begin{aligned}
\text{step}((\mathbf{1} \,\|\, (\mathbf{1} \,;\, \mathbf{1})), 1) &= \Big\{(P' \,\|\, Q') \colon P' \in \text{step}(\mathbf{1}, m_P), Q' \in \text{step}((\mathbf{1} \,;\, \mathbf{1}), m_Q), \\
&\qquad m_P \in [0,1], m_Q \in [0,1], m_P + m_Q = \min\{2,1\}\Big\}
\end{aligned}$$

which means that either $m_P = 1, m_Q = 0$ or $m_P = 0, m_Q = 1$. Taking the union of both alternatives results in

$$\begin{aligned}
&= \Big\{(P' \,\|\, Q') \colon P' \in \text{step}(\mathbf{1}, 1), Q' \in \text{step}((\mathbf{1} \,;\, \mathbf{1}), 0)\Big\} \\
&\qquad \bigcup \Big\{(P' \,\|\, Q') \colon P' \in \text{step}(\mathbf{1}, 0), Q' \in \text{step}((\mathbf{1} \,;\, \mathbf{1}), 1)\Big\} \\
&= \Big\{(P' \,\|\, Q') \colon P' \in \{\mathbf{0}\}, Q' \in \{\mathbf{1} \,;\, \mathbf{1}\}\Big\} \\
&\qquad \bigcup \Big\{(P' \,\|\, Q') \colon P' \in \{\mathbf{1}\}, Q' \in \{\mathbf{1}\}\Big\} \\
&= \{\mathbf{0} \,\|\, (\mathbf{1} \,;\, \mathbf{1})\} \bigcup \{\mathbf{1} \,\|\, \mathbf{1}\} \\
&= \{(\mathbf{1} \,;\, \mathbf{1}), (\mathbf{1} \,\|\, \mathbf{1})\}
\end{aligned}$$

The result set has more than one element. Therefore, the result of the first step of P is not deterministic, but depends on which branch is assigned the one available processor.

7.2.4 Schedules

A schedule is a sequence of the number of processors available to be assigned at consecutive points in time, modeled as a finite sequence of non-negative integers. The set of all schedules is denoted $\mathbb{S}$. Bracket notation will be used for sequences, eg, $\langle 1, 2, 3 \rangle$ for 1 followed by 2 followed by 3; and $\langle\rangle$ for the empty sequence. The concatenation of two sequences is written $s_1 \frown s_2$, eg,

$$\langle 1, 2, 3 \rangle \frown \langle 4, 5, 6 \rangle = \langle 1, 2, 3, 4, 5, 6 \rangle$$

The results of executing a process P on a schedule s will be expressed by the operator $\otimes \colon \mathbb{P} \times \mathbb{S} \to \{\mathbb{P}\}$, defined by

$$\begin{aligned} P \otimes \langle\rangle &= \{P\} \\ P \otimes (\langle m \rangle \frown s) &= \bigcup_{P' \in \mathsf{step}(P,m)} P' \otimes s \end{aligned} \tag{7.6}$$

A process P *will complete* on schedule s if it is guaranteed to complete, ie,

$$P \otimes s = \{\mathbf{0}\} \tag{7.7}$$

This is distinct from *may complete*, as in

$$\mathbf{0} \in P \otimes s \tag{7.8}$$

Example 7.2.3
Let P be a process, defined by

$$P = (\mathbf{1} \,;\, \mathbf{1}) \,\|\, \mathbf{1} \,\|\, \mathbf{1} \tag{7.9}$$

Looking at P, one can see that there are three possible $\mathbf{1}$-processes that can be executed at the first step. This is equivalent with the fact that $\mathcal{H}(P) = 3$. Also, no matter how many processors that are given to P it will never complete in less than two steps due to the sequence process in the left branch. This is equivalent with $\mathcal{L}(P) = 2$. The total number of $\mathbf{1}$-processes in P is four, so $\mathcal{C}(P) = 4$.

Consider the schedule $s = \langle 2, 3 \rangle$, meaning that P is given two processors for its first step, and three processors for its second step. Will P complete? Beginning with $\mathsf{step}(P, 2)$, we get

$$\mathsf{step}(P, 2) = \{(\mathbf{1} \,;\, \mathbf{1}), (\mathbf{1} \,\|\, \mathbf{1})\}$$

For each process P' of the resulting set, we apply $\mathsf{step}(P', 3)$:

$$\begin{aligned} \mathsf{step}((\mathbf{1} \,;\, \mathbf{1}), 3) &= \{\mathbf{1}\} \\ \mathsf{step}((\mathbf{1} \,\|\, \mathbf{1}), 3) &= \{\mathbf{0}\} \end{aligned}$$

Therefore,

$$P \otimes s = \{\mathbf{0}, \mathbf{1}\}$$

so P may or may not complete on the schedule.

7.3 Partial Orders on CTPs

In order to analyze CTPs, some means of comparing processes must be introduced. In this section, two partial orders will be defined that allow comparison between processes with respect to different properties. The first is the upper bound order, which relates to worst-case execution time. A process Q is said to be an upper bound of P if P has the same structure as Q, but with possibly some of the computation removed. The second is the schedulability order, relating to the ease of scheduling a process. A process P is said to be easier to schedule than Q, if Q always completing on a schedule implies that P will also always complete.

7.3.1 Upper Bound Order ($\sqsupseteq$)

Execution time estimates are upper bounds, so there is always the chance that an execution of a program turns out to require less computation than a CTP based on execution time analysis of the program. If a process P represents such a possible execution of a process Q, then Q is said to be an upper bound for P. This will be written $P \sqsupseteq Q$.

Definition 7.1 ($\sqsupseteq$)
Q is an *upper bound* of P, written $P \sqsupseteq Q$ if P can be derived from Q by replacing any number of unit processes in Q with the zero process.

It follows per definition that

$$\begin{aligned} &\mathbf{0} \sqsupseteq \mathbf{1} && \\ &P' \,;\, Q' \sqsupseteq P \,;\, Q && \text{if } P' \sqsupseteq P \wedge Q' \sqsupseteq Q \\ &P' \,\|\, Q' \sqsupseteq P \,\|\, Q && \text{if } P' \sqsupseteq P \wedge Q' \sqsupseteq Q \end{aligned}$$

The upper bound relation satisfies all properties of a partial order:

$$\begin{aligned} P \sqsupseteq P & & \text{(reflexive)} \\ P \sqsupseteq Q \wedge Q \sqsupseteq P &\implies P = Q & \text{(anti-symmetric)} \\ P \sqsupseteq Q \wedge Q \sqsupseteq R &\implies P \sqsupseteq R & \text{(transitive)} \end{aligned}$$

The relation is not a total order, in that there exist pairs of processes where neither is an upper bound of the other, for example $\mathbf{1}\,;\mathbf{1}$ and $\mathbf{1}\,\|\,\mathbf{1}$. An important property is that executing a process can be seen as a special case of removing computation:

$$\forall P \in \mathbb{P} \colon \forall s \in \mathbb{S} \colon (P' \in P \otimes s \implies P' \sqsupseteq P) \tag{7.10}$$

7.3.2 Schedulability Order (≤)

The schedulability order relates two processes P and Q in such a way that if one process is guaranteed to complete on a schedule, then the other is also guaranteed to complete on that schedule.

Definition 7.2 (≤)
A process P is *easier to schedule* than a process Q, written $P \leq Q$, if for all schedules s,

$$Q \otimes s = \{\mathbf{0}\} \implies P \otimes s = \{\mathbf{0}\} \tag{7.11}$$

This order is essentially an order of worst-case execution time, as it relates the worst-case completion of P to the worst-case completion of Q. If $P \leq Q$, and Q is known to complete on some schedule, then Q can be replaced with P, and P will also complete on that schedule. Trivially,

$$\mathbf{0} \leq \mathbf{1}$$

because $\mathbf{0}$ is guaranteed to complete on any schedule. The ≤-relation also satisfies all properties of a partial order:

$$\begin{aligned} &P \leq P && \text{(reflexive)} \\ P \leq Q \wedge Q \leq P &\implies P = Q && \text{(anti-symmetric)} \\ P \leq Q \wedge Q \leq R &\implies P \leq R && \text{(transitive)} \end{aligned}$$

Like the ⊒-relation, the ≤-relation is also not a total order. For example, if

$$\begin{aligned} P &= \mathbf{1}\,\|\,\mathbf{1}\,\|\,\mathbf{1} & s_1 &= \langle 3 \rangle \\ Q &= \mathbf{1}\,;\mathbf{1} & s_2 &= \langle 1, 1 \rangle \end{aligned}$$

Then P and Q are incomparable with respect to schedulability. P will always complete for s_1 but never for s_2, while Q will always complete for s_2 but never for s_1.

Unlike the ⊒-relation, the ≤-relation does not in general distribute over "; " and "$\|$", but it does distribute for the right element of a sequence, as maintained by the following lemma:

Lemma 7.1. *For all processes P and Q, and all processes Q' where $Q' \leq Q$,*

$$P\,;\,Q' \leq P\,;\,Q \tag{7.12}$$

Proof. Proof by contradiction. Assuming that eq. (7.12) does not hold, then there must exist a schedule s so that $(P\,;\,Q) \otimes s = \{\mathbf{0}\}$ and $(P\,;\,Q') \otimes s \neq \{\mathbf{0}\}$. For all executions of $P\,;\,Q$ on this schedule there exists some prefix of s so that

$$s = s_P \frown s_Q.$$
$$Q \in (P\,;\,Q) \otimes s_P$$

As $P\,;\,Q$ will always complete on s it follows that $Q \otimes s_Q = \{\mathbf{0}\}$. However $Q' \leq Q$ implies $Q' \otimes s_q = \{\mathbf{0}\}$, so $P\,;\,Q'$ must also always complete, contradicting the assumption that $P\,;\,Q'$ did not complete on the schedule. □

7.4 Ill-Behaved Processes

Say a process Q has been derived from execution time analysis of some program. Because the WCET analysis always yields worst-case timings, an actual instance of the program does not necessarily behave like Q, but is only guaranteed to behave like some process P for which Q is an upper bound. Consider the following implication:

$$P \sqsupseteq Q \overset{?}{\Longrightarrow} P \leq Q \tag{7.13}$$

If the above statement was true, then an execution of the program would always have behaved like Q or like some process easier to schedule than Q: If Q was schedulable, then the program would always be schedulable. However, and somewhat surprisingly, the above statement does generally not hold. A special case of this, which may seem even more counterintuitive, is illustrated by the following observation:

Observation 7.1. *There exist processes P and Q, and a schedule s so that*

$$(P \in Q \otimes s) \wedge (P \nleq Q) \tag{7.14}$$

that is, there may be situations where Q is unable to complete on a schedule *if and only if* it has completed some execution prior to beginning on the schedule.

Example 7.4.1
An example will be given. Let P, Q and s be defined by

$$\begin{aligned} Q &= (\mathbf{1}\,;(\mathbf{1}\,\|\,\mathbf{1}))\;\|\;(\mathbf{1}\,;(\mathbf{1}\,\|\,\mathbf{1})) \\ P &= (\mathbf{1}\,;(\mathbf{1}\,\|\,\mathbf{1}))\;\|\;(\mathbf{1}\,\|\,\mathbf{1}) \\ s &= \langle 1\rangle \end{aligned} \tag{7.15}$$

By choosing to execute the right parallel branch of Q, one can see that $P \in Q \otimes s$. To show $P \not\leq Q$ it is sufficient to find a schedule for which Q is guaranteed to complete, but P is not. Let u be the schedule defined by

$$u = \langle 2, 4\rangle$$

By computing $Q \otimes u$ one finds that Q is guaranteed to complete on u:

$$\begin{gathered} \big((\mathbf{1}\,;(\mathbf{1}\,\|\,\mathbf{1}))\;\|\;(\mathbf{1}\,;(\mathbf{1}\,\|\,\mathbf{1}))\big)\otimes\langle 2\rangle = \{\mathbf{1}\,\|\,\mathbf{1}\,\|\,\mathbf{1}\,\|\,\mathbf{1}\} \\ (\mathbf{1}\,\|\,\mathbf{1}\,\|\,\mathbf{1}\,\|\,\mathbf{1})\otimes\langle 4\rangle = \{\mathbf{0}\} \end{gathered}$$

By computing $P \otimes u$ one finds that P is not guaranteed to complete on u. The first step yields

$$\big((\mathbf{1}\,;(\mathbf{1}\,\|\,\mathbf{1}))\;\|\;(\mathbf{1}\,\|\,\mathbf{1})\big)\otimes\langle 2\rangle = \big\{(\mathbf{1}\,\|\,\mathbf{1}\,\|\,\mathbf{1}),\ (\mathbf{1}\,;(\mathbf{1}\,\|\,\mathbf{1}))\big\}$$

Computing the second step for each of these results yields

$$\begin{gathered} (\mathbf{1}\,\|\,\mathbf{1}\,\|\,\mathbf{1})\otimes\langle 4\rangle = \{\mathbf{0}\} \\ (\mathbf{1}\,;(\mathbf{1}\,\|\,\mathbf{1}))\otimes\langle 4\rangle = \{\mathbf{1}\,\|\,\mathbf{1}\} \end{gathered}$$

As Q will complete on u, but P may or may not, $P \not\leq Q$.

Corollary 7.2. *There exist processes $Q \in \mathbb{P}$ and schedules $u, v \in \mathbb{S}$, where $\forall i\colon u_i \leq v_i$, and where*

$$(Q \otimes u = \{\mathbf{0}\}) \wedge (Q \otimes v \neq \{\mathbf{0}\}) \tag{7.16}$$

Proof. This can be shown setting $u = \langle 0, 2, 4\rangle$ and $v = \langle 1, 2, 4\rangle$ and using P and Q from example 7.4.1. Then, $Q \otimes \langle 0\rangle = \{Q\}$ and $Q \otimes \langle 1\rangle = \{P\}$. The rest of the schedule is $\langle 2, 4\rangle$, for which Q will always complete, but P may not. □

Corollary 7.3. *There exist processes P and Q so that*

$$P \sqsupseteq Q \wedge P \not\leq Q \tag{7.17}$$

Proof. Executing a process will always result in a new process for which the original is an upper bound (see eq. (7.10)). The processes in example 7.4.1 therefore satisfies $P \sqsupseteq Q$, and $P \npreceq Q$. □

For a process $P \sqsupseteq Q$ not to complete when Q is guaranteed to complete, it is necessary that the scheduler at some point makes a "wrong" decision. An argument may therefore be made that the counterintuitive behavior of processes is in some part due to the scheduler; and moreover, that this behavior can be eliminated by attempting to make a better scheduling algorithm. The following observation illustrates the futility of such an attempt.

Observation 7.2. *There exists some process Q and schedule s so that the set $Q \otimes s$ has no least element in the schedulability order.*

Example 7.4.2
An example will be given. Consider the schedule $s = \langle 1, 3\rangle$ and the process Q defined by

$$Q = (\mathbf{1}\,;(\mathbf{1}\,\|\,\mathbf{1}))\ \|\ (\mathbf{1}\,;\,\mathbf{1}\,;\,\mathbf{1})$$

The first step of computation results in the following two processes:

$$\begin{aligned} Q \otimes \langle 1\rangle = \Big\{&(\mathbf{1}\,\|\,\mathbf{1}\,\|\,(\mathbf{1}\,;\,\mathbf{1}\,;\,\mathbf{1})),\\ &(\mathbf{1}\,;(\mathbf{1}\,\|\,\mathbf{1}))\ \|\ (\mathbf{1}\,;\,\mathbf{1})\Big\}\end{aligned}$$

Computing the second step for each of these processes yields

$$\begin{aligned} \big(\mathbf{1}\,\|\,\mathbf{1}\,\|\,(\mathbf{1}\,;\,\mathbf{1}\,;\,\mathbf{1})\big)\otimes\langle 3\rangle &= \{\mathbf{1}\,;\,\mathbf{1}\}\\ \big((\mathbf{1}\,;(\mathbf{1}\,\|\,\mathbf{1}))\ \|\ (\mathbf{1}\,;\,\mathbf{1})\big)\otimes\langle 3\rangle &= \{\mathbf{1}\,\|\,\mathbf{1}\,\|\,\mathbf{1}\}\end{aligned}$$

so

$$Q \otimes s = \Big\{(\mathbf{1}\,;\,\mathbf{1}),\ (\mathbf{1}\,\|\,\mathbf{1}\,\|\,\mathbf{1})\Big\} \tag{7.18}$$

The two resulting processes are incomparable with respect to schedulability, so $Q \otimes s$ has no least element.

Observation 7.2 implies that there is generally no "correct choice" for a scheduler; whether or not a scheduling choice leads to the completion of a process may depend on the future schedule. The future schedule again depends on the behavior of other tasks in the system, and is generally not predictable. For example, take the processes in eq. (7.18). If the rest of the schedule is $\langle 1, 1\rangle$, then the first result will complete the process, but not the second. If a future schedule is $\langle 3\rangle$, then the second result completes the process, but not the first.

7.5 Well-behaved Processes

In the previous section it was demonstrated that there exist processes which are harder to schedule if computation is removed from them. Such a process should be avoided when analyzing schedulability, because for real programs, only upper bounds on computation may be determined in advance. However, some processes do not exhibit this kind of behavior. These processes will be referred to as well-behaved, and will be the main topic of this section.

Definition 7.3 (Well-behaved)
A computation time process Q is well-behaved if and only if

$$\forall P \in \mathbb{P} \colon (P \sqsupseteq Q \implies P \leq Q) \tag{7.19}$$

Whether or not a process Q is well-behaved can be determined by exhaustive examination of all schedules in which Q is guaranteed to complete, and checking these schedules against all processes $P \sqsupseteq Q$ for which Q is an upper bound. However, such a test has at least exponential complexity with respect to $C(Q)$, making it infeasible for most practical purposes. Instead, one may try to find general classes of processes where good behavior is guaranteed by the process structure.

7.5.1 Classes of Well-Behaved Processes

Some simple processes are quickly found to be well-behaved:

Lemma 7.4. *The zero process ($\mathbf{0}$) and the unit process ($\mathbf{1}$) are well-behaved.*

Proof. The only process for which $\mathbf{0}$ is an upper bound is $\mathbf{0}$ itself, so it follows that $\mathbf{0}$ is well-behaved. The process $\mathbf{1}$ will complete on any schedule with at least one non-zero element. The only processes for which $\mathbf{1}$ is an upper bound is $\mathbf{0}$ and $\mathbf{1}$. Both will complete on any non-zero schedule, so it follows that $\mathbf{1}$ is well-behaved. □

Theorem 7.5. *Let P be a process with the following structure:*

$$P = P_1 \,;\, P_2 \,;\, P_3 \,;\, \ldots \,;\, P_n$$

If the processes $P_i|_{i=1\ldots n}$ are well-behaved, then P is well-behaved.

Proof. Note that it is sufficient to prove that $P_1 \,;\, P_2$ is well-behaved. If that is true, then

$$P = (P_1 \,;\, P_2) \,;\, P_3 \,;\, \ldots \,;\, P_n \tag{7.20}$$

will be a sequence of well-behaved processes for which the first element, $(P_1 \,;\, P_2)$, is well-behaved. The same proof can then be used to show that $((P_1 \,;\, P_2) \,;\, P_3)$ is well-behaved, and so on.

Let $s \in \mathbb{S}$ be any schedule for which $(P_1 \,;\, P_2) \otimes s = \{\mathbf{0}\}$. Let s_1 be the shortest prefix of s for which $P_1 \otimes s_1 = \{\mathbf{0}\}$ and let s_2 be the suffix of s so that $s = s_1 ^\frown s_2$. It follows that $P_2 \otimes s_2 = \{\mathbf{0}\}$, otherwise it would be possible to execute $(P_1 \,;\, P_2)$ on s without it completing. Let P_1' and P_2' be processes so that $P_1' \sqsupseteq P_1$ and $P_2' \sqsupseteq P_2$. Because P_1 is well-behaved, $P_1' \otimes s_1 = \{\mathbf{0}\}$.

P_1' must complete for s_1, but it may also complete earlier, leaving some rest of the schedule s_{rest}. It remains to show that all processes P_2' must complete within the schedule $s_{\text{rest}} ^\frown s_2$. As was noted in eq. (7.10), execution of a process must lead to a process for which the original is an upper bound. Therefore, for all s_{rest},

$$\forall P_2'' \in P_2' \otimes s_{\text{rest}} \colon P_2'' \sqsupseteq P_2' \sqsupseteq P_2$$

The remaining schedule is s_2. It was already shown that $P_2 \otimes s_2 = \{\mathbf{0}\}$. As P_2 is well behaved, and $P_2'' \sqsupseteq P_2$, then P_2'' will also always complete on s_2. This shows that $P_1' \,;\, P_2' \leq P_1 \,;\, P_2$, so $P_1 \,;\, P_2$ is well-behaved. □

In general, $P \,\|\, Q$ is not well-behaved, even if P and Q are well-behaved. For example, the Q given in eq. (7.15) is not well-behaved even though both of the parallel branches are well-behaved. A special case where a parallel process is indeed well-behaved is given below:

Theorem 7.6. *If $Q \in \mathbb{P}$ has the following structure*

$$Q = (\mathbf{1} \,;\, \mathbf{1} \,;\, \ldots \,;\, \mathbf{1}) \,\|\, (\mathbf{1} \,;\, \mathbf{1} \,;\, \ldots \,;\, \mathbf{1}) \,\|\, \ldots \tag{7.21}$$

then Q is well-behaved.

Proof. First note that all processes for which Q is an upper bound has the same structure as Q.

Let $Q = Q_1 \,\|\, Q_2 \,\|\, \ldots Q_N$ so that the Q_i processes are sequences of $\mathbf{1}$s. If some P satisfies $P \sqsupseteq Q$, then P must be equal to Q with some $\mathbf{1}$s removed, so P can be written as $P = P_1 \,\|\, P_2 \,\|\, \ldots P_N$, in such a way that

$$\forall i \in [1, N] \colon \mathcal{L}(P_i) \leq \mathcal{L}(Q_i) \tag{7.22}$$

We also have $\mathcal{C}(P) \leq \mathcal{C}(Q)$ and $\mathcal{H}(P) \leq \mathcal{H}(Q)$. Let m be the first element in a schedule. If

$$m \leq \mathcal{H}(P) \;\vee\; m \geq \mathcal{H}(Q)$$

then every choice that the scheduler can make for P it can also make for Q. For all choices of P, a corresponding choice can be made for Q so that eq. (7.22) is still satisfied. If

$$\mathcal{H}(P) \leq m \leq \mathcal{H}(Q)$$

then the scheduler may choose to execute additional branches for Q. However, these branches must already be of zero length for P, so for all results for P, a corresponding result for Q can be found that satisfies eq. (7.22). This can be repeated for each element in the schedule.

This shows that for all schedules s, and all processes $P' \in P \otimes s$ there exists some process $Q' \in Q \otimes s$ so that $C(P') \leq C(Q')$. A consequence is that we cannot have $Q \otimes s = \{\mathbf{0}\}$ when $P \otimes s \neq \{\mathbf{0}\}$. Therefore,

$$Q \otimes s = \{\mathbf{0}\} \implies P \otimes s = \{\mathbf{0}\}$$

P is easier to schedule than Q, which implies that Q is well-behaved. □

7.5.2 Safe Upper Bound

If schedulability analysis is to be performed on some ill-behaved process P, then the analysis will not be sustainable. To make the analysis sustainable, it would be better to replace P with some well-behaved process Q that is harder to schedule than all processes $P' \sqsupseteq P$, and then analyze Q instead. Such a Q will be referred to as a safe upper bound.

Definition 7.4 (Safe Upper Bound)
A process Q is a safe upper bound for a process P if

$$\forall P' \sqsupseteq P \colon P' \leq Q \tag{7.23}$$

The existence of a safe upper bound is guaranteed by the following lemma:

Lemma 7.7. *For all $P \in \mathbb{P}$, if Q is a sequence of $\mathbf{1}$s with length $C(P)$, then Q is a safe upper bound for P.*

Proof. Q will complete for all schedules that have at least $C(P)$ non-zero elements. P will also complete for all these schedules, so $P \leq Q$. Furthermore, all processes P' that satisfy $P' \sqsupseteq P$ will also complete for this schedule, so Q is a safe upper bound. □

A process P can be replaced by a safe upper bound to make temporal analysis of P sustainable. However, the analysis will no longer be exact, as the safe upper bound may be harder to schedule than P. For example, the choice of safe upper bound used in lemma 7.7 suppresses all parallelism of a process, which could make the analysis unnecessarily pessimistic. Sometimes, other safe upper bounds exist that maintain some of this parallelism and thus lead to less pessimistic results:

Example 7.5.1
Take the ill-behaved process Q given below:

$$Q = (\mathbf{1}\,;(\mathbf{1}\,\|\,\mathbf{1}))\;\|\;(\mathbf{1}\,;(\mathbf{1}\,\|\,\mathbf{1}))$$

Let Q_S be defined by

$$Q_S = \mathbf{1}\,;\,\mathbf{1}\,;\,\mathbf{1}\,;\,\mathbf{1}\,;\,\mathbf{1}\,;\,\mathbf{1}$$

According to lemma 7.7, Q_S is a safe upper bound for Q. Let Q_P be defined by

$$Q_P = (\mathbf{1}\,\|\,\mathbf{1})\,;(\mathbf{1}\,\|\,\mathbf{1})\,;(\mathbf{1}\,\|\,\mathbf{1}) \tag{7.24}$$

Q_P will complete on some schedule s if and only if $s = b_1 \frown b_2 \frown b_3$ where each $b_i|_{i=1..3}$ is either $\langle 1, 1\rangle$ or $\langle 2\rangle$, or similar schedules with strictly larger elements. For example, Q_P will complete on $\langle 2, 1, 1, 2\rangle$, but not on $\langle 1, 2, 1, 2\rangle$. It can be found by systematic examination that Q will complete on all these schedules, and that all $Q' \sqsupseteq Q$ also will complete on these schedules. We know from theorem 7.5 that Q is well-behaved. It follows that Q_P is also a safe upper bound of Q. Moreover, as $Q_P \leq Q_S$ there are no schedules for which Q_S will complete and Q_P will not. Q_P is therefore a better choice of safe upper bound.

For a process P, the best choice of safe upper bound would be the least element in the set of safe upper bounds with respect to schedulability:

$$Q^\star = \min_{\leq}\{Q \in \mathbb{P} \colon \forall P' \sqsupseteq P \colon P' \leq Q\} \tag{7.25}$$

At this point no method for finding the best safe upper bound is known, or if a best safe upper bound generally exists.

7.6 Discussion

This chapter introduced the computation time process model as a tool for temporal analysis of non-communicating programs with an arbitrary parallel structure. The simplicity of dealing only with time, rather than with time and computation, allows the temporal properties of simple processes to be examined in isolation more easily than would be possible with general timed process algebras such as Timed CSP [86]. Moreover, the CTP model explicitly models timing of programs on multiprocessor systems, not timing of abstract processes in general.

Somewhat counter-intuitively it was shown that a worst-case estimate of execution time does not in general represent a worst-case scenario with respect to schedulability; there exist processes that are unable to complete on schedule only if requiring less than their WCET. The existence of these ill-behaved processes has important implications for temporal analysis of programs with an arbitrary **SEQ/PAR**-structure, because it implies that exact schedulability analysis will in general not be sustainable.

In practical systems there are several ways to alleviate the problem caused by ill-behaved processes. One way is to avoid them altogether, and require that all jobs to be analyzed have one of the parallel structures guaranteed not to be ill-behaved. Another way is to perform the analysis using safe upper bounds, although this would require the development of a method for systematically deriving good, safe upper bounds.

Finally, it is also possible to avoid these problems by using a non-work-conserving scheduler. The most straightforward way would be to require that each parallel branch always use up its maximum execution time, by spending any remaining CPU cycles doing nothing. However, this solution has multiple disadvantages, including worse average performance and increased power consumption, and is also likely to add a fair bit of overhead to the intra-job scheduler.

In the next chapter, an analysis is developed under the assumption that the intra-job scheduler is reasonably fair, and not undefined, as in this chapter. This does not eliminate the existence of ill-behaved processes, but it allows a simple method of finding sustainable upper bounds on execution times even for processes that are ill-behaved.

Chapter 8

Analysis of Job-Level Parallelism with a Fair Intra-job Scheduler

Hospitals Sued by 7 Foot Doctors

Newspaper headline

This chapter presents a framework for analyzing complete systems with malleable jobs of an arbitrary parallel structure. However, unlike in chapter 7, where the intra-job scheduler was undefined, the analysis in this chapter requires the intra-job scheduler to be reasonably fair: It must always distribute the processors available to a job evenly amongst the parallel branches of a job.

8.1 Introduction

The fair intra-job scheduler allows the progress of a job to be represented by a scalar and its parallel structure to be modeled as a function. Together with the assumption of continuous time, this enables the use of real-valued mathematical analysis, making it significantly easier to reason on the schedulability of complete systems. However, as both real-valued time and the fair scheduler are idealizations that are not perfectly realizable, this change introduces errors that the analysis must explicitly take into account.

It is demonstrated that even under a fast intra-job scheduler, jobs requiring their worst-case execution times do not necessarily constitute a worst-case scenario with respect to schedulability. This implies that exact schedulability analysis for this framework cannot be sustainable.

Upper bounds on interference and demand are developed for the new model. The resulting framework is used to construct a pessimistic, but sustainable schedulability test for systems scheduled with EDF. The EDF test has poor worst-case performance, but does allow schedulability analysis for a class of systems for which no other analysis currently exists.

8.1.1 Outline

Section 8.2 describes the job model and the intra-job scheduling model. Section 8.3 discusses schedulability, WCETs and sustainability of schedulability analysis for the tasking model of this chapter. Section 8.4 derives a sufficient condition for when a job meets its deadline, based on the interference from other jobs and an upper bound on the skew (unfairness) of the intra-job scheduler. An upper bound on the interference to a job as a function of the demand of higher priority jobs is also derived, as well as an upper bound on the demand of a job for a given window in time. In section 8.5 these bounds are combined to construct a schedulability analysis for systems scheduled with EDF.

8.2 System Model

The idea behind the fair intra-job scheduler is to allow all branches of a job to progress at the same speed, so that the progress of a job can be represented by a single value v, which will be called the virtual time. The parallel structure of a job can then be represented by a function $c(v)$, which will be called the computation function of the job.

8.2.1 Basic Assumptions

Jobs will be assumed to have a **SEQ/PAR**-structure. It will be assumed that there is no other synchronization within a job, apart from the implicit synchronization imposed by the sequence operator. The pseudo-statement "Exec(d)" will denote some computation that requires d units of serial execution. Two jobs will be said to have the same structure iff they can be written so that they only differ in the values of their Exec statements.

The EDF analysis will require constrained deadlines, ie, $D_A \leq T_A$ for all tasks. Furthermore, an ideal system model will be assumed, with no jitter and no cost of preemptions. It will also be assumed that jobs are independent.

8.2.2 Computation Functions and Virtual Time

The fast intra-job scheduler allows the **SEQ/PAR**-structure of a job to be represented by an abstract function which is called the computation function of the job:

Definition 8.1 (Computation Function)
The computation function $c(v)$ of a job denotes the maximum number of processors that the job can utilize at virtual time v for that job.

The computation function of a job is equivalent to the number of processors used by a job as a function of time, if the job is always given as many processors as it can utilize.

For $c(v)$ to be a valid computation function there must exist a L so that

$$\begin{aligned} c(v) > 0 &\quad \text{when } 0 \leq v \leq L \\ c(v) = 0 &\quad \text{otherwise} \end{aligned} \tag{8.1}$$

The parameter L is the minimum completion time, or length, of the job, and is the time required for a job to complete if it is always given as many processors as it can utilize. The total computational requirement of a job, C, can then be defined as a function of $c(v)$:

$$C = \int_0^\infty c(v)\,\mathrm{d}v = \int_0^L c(v)\,\mathrm{d}v \tag{8.2}$$

Example 8.2.1
Let a, b and c denote the jobs given in listings 8.1 to 8.3, respectively. These jobs have the computation functions shown in figs. 8.1a to 8.1c. Their minimum completion times are $L_a = 2$, $L_b = 2$ and $L_c = 3$, and their total computation is $C_a = 6$, $C_b = 5$ and $C_c = 6$.

Definition 8.2 (Virtual Time)
The virtual time of a job is denoted $v(t)$ and is the amount of time required to reach the current point of execution if the job is always given as many processors as it can utilize. $v(t)$ is described by the following equations:

$$\begin{aligned} v(r) &= 0 \\ \frac{\mathrm{d}}{\mathrm{d}t} v(t) &= \min\left\{\frac{s(t)}{c(v(t))}, 1\right\} \end{aligned} \tag{8.3}$$

where r is the release time of the job, and $s(t)$ is the *schedule*, ie, the number of processors available for the job at time t.

Listings 8.1 to 8.3: *Structure of Jobs a, b and c in Example 8.2.1*

```
-- a
PAR
  SEQ
    Exec(1)
    PAR
      Exec(1)
      Exec(1)
  SEQ
    Exec(1)
    PAR
      Exec(1)
      Exec(1)
:
```

```
-- b
PAR
  SEQ
    Exec(1)
    PAR
      Exec(1)
      Exec(1)
  Exec(1)
  Exec(1)
:
```

```
-- c
PAR
  Exec(3)
  SEQ
    Exec(1)
    PAR
      Exec(1)
      Exec(1)
:
```

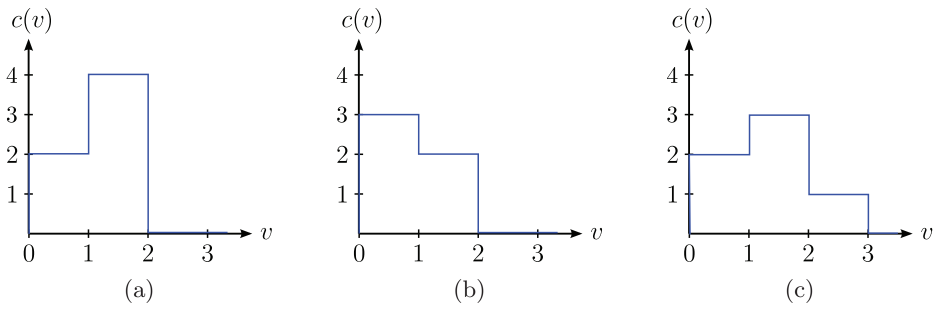


Figure 8.1: Computation functions of (a) job in listing 8.1, (b) job in listing 8.2, (c) job in listing 8.3.

For completeness, we also define $\frac{\mathrm{d}}{\mathrm{d}t}v(t) = 0$ when $c(v(t)) = 0$, ie, the job is defined to have zero progress before it begins and after it completes.

The execution of a job is complete when $v(t) = L$, since there is no more remaining computation. Virtual time follows real-time whenever a task is allowed as many processors as it can utilize, and slows down when fewer processors are available.

For all t, $v(t)$ is continuous and monotonically increasing.

8.2.3 Skew

The fair intra-job scheduler is an idealization that is not perfectly realizable in a real system. If the scheduler allows some branches to advance more rapidly than others then this will be called *skew*.

When the intra-job scheduler is perfectly fair, a job that is given some CPU will always decrease its remaining minimum completion time, but when the scheduler is skewed this is no longer the case. For example, if the scheduler does not execute the longest branch of the job, then the remaining minimum completion time will stay the same.

The skew is assumed to be bounded by a number σ, such that for any window of length l, one branch of a job is allowed to execute at most $l \cdot \sigma$ more than any other branch. This means that for a job a that meets its deadline, skew in the intra-job scheduler will contribute at most $\sigma \cdot D_a$ to the completion time of a.

8.2.4 Reducing Computation Functions

The computation function of a job is independent of the number of processors in the system (m), and only depends on the structure and execution times of the parallel branches of the job. However, no job will ever be given more than m processors simultaneously, and to simplify some of the later analysis it is convenient that $c(v) \leq m$ for all v. A *reduced computation function* is a function where $c(v) \leq m$ for all v, and which behaves equivalently with the original function in systems of m processors.

A reduced computation function can be interpreted as the number of processors used by the job as a function of time since the release of the job, given that there are no other jobs in the system. As long as computation functions are reduced, virtual time will follow real time when there is no interference from other jobs.

A computation function $c(v)$ can be reduced to m processors by computing

$$c_{(m)}(t) = \min\{c(v(t)), m\} \tag{8.4}$$

using eq. (8.3) with $s(t) = m$. Informally, a reduced computation function is found by limiting all sections of the function where $c(t) > m$ to m, and extending them by a factor of $c(t)/m$, conserving the total amount of computation in the section.

8.3 Sustainability

When analyzing schedulability of systems that allow arbitrary structured JLP, certain scheduling analysis concepts must be redefined in order to discuss sustainability. Consider the following questions: If one compares two alternative jobs a and b, with identical releases and deadlines, what does it mean that

1. the computation required by b is an upper bound to the computation required by a,
2. a is easier to schedule than b,
3. b has a higher demand than a, and thus may cause more interference to lower priority jobs.

For systems without JLP, these statements are all equivalent to $C_a \leq C_b$.

In the last chapter it was shown that answers to questions 1 and 2 might vary independently for jobs when the intra-job scheduler was undefined. Question 3 was not discussed. In this section it will be shown that all these questions are independent under a fair intra-job scheduler.

8.3.1 Definitions

First, consider the concept of upper bounds on computation.

Definition 8.3 (Upper Bound on Computation)
For a job b, if there exists a job a that has the same structure as b, but with a less or equal amount of computation, then b is said to be an upper bound of a. This is written $a \sqsupseteq b$.

Computation times derived from WCET analysis of actual programs are worst-case estimates. Therefore, in an actual execution of a job, any or all of the branches in the job may require less than their expected execution time. A consequence is that even if execution time analysis of a job b leads to a computation function of $c_b(v)$, the only thing known about an actual execution of b is that it will behave as if having the computation function $c_a(v)$ of some job a for which b is an upper bound.

Second, consider what it means for a job to be easier to schedule than another.

Definition 8.4 (Easier to schedule)
For two jobs a and b, a is easier to schedule than b, if and only if for all schedules

$$\forall t\colon v_a(t) = L_a \implies v_b(t) = L_b \tag{8.5}$$

In other words, a is easier to schedule than b, if b meeting its deadline in some schedule implies that a will also meet its deadline given the same schedule.

Third, consider the demand of a job.

Definition 8.5 (Demand)
The demand of a job a in a window $[t_0, t_1]$ is denoted $\mathcal{D}_a(t_0, t_1)$ and is given by

$$\mathcal{D}_a(t_0, t_1) = \int_{v(t_0)}^{v(t_1)} c_a(v')\,\mathrm{d}v' \tag{8.6}$$

The demand of a job is the total amount of execution performed by the job in the given window, and is dependent on the schedule.

8.3.2 Non-Sustainability of Exact Analysis

The fair intra-job scheduler does not eliminate the temporal anomalies described in chapter 7, where a job would complete on a given schedule if executing its full worst-case computation, but may no longer complete if executing less:

Example 8.3.1
Let a and b be the jobs in listings 8.1 and 8.2, respectively. Because b has the same structure as a, but with less computation, then $b \sqsupseteq a$.

Let the jobs both be given the schedule depicted in fig. 8.2c. Virtual time when executing the two jobs is shown in figs. 8.2a and 8.2b. Job a completes at $t = 2$. Although a is an upper bound of b, b does not complete until $t \approx 2.33$. If $c_a(v)$ was a WCET estimate of $c_b(v)$, and the deadline was 2, then the job would meet its deadline if executing its full worst-case computation, but may miss its deadline if executing less.

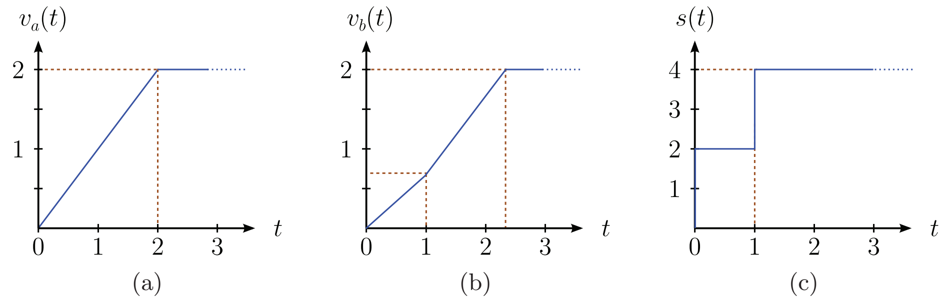


Figure 8.2: Schedule and virtual times for Example 8.3.1. (a) shows virtual time for job a; (b) for job b. (c) shows schedule for both jobs.

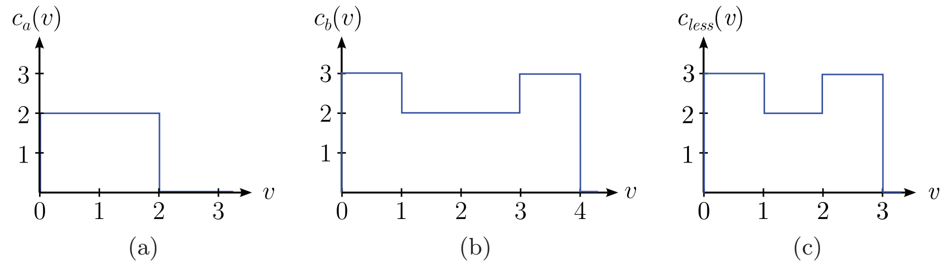


Figure 8.3: *$c_a(v)$, $c_b(v)$ and $c_{\text{less}}(v)$ for Example 8.3.2*

A similar anomaly may occur due to interference to a low priority job from a high priority job. This is illustrated in the following observation:

Observation 8.1. *Lower priority jobs may miss their deadlines if and only if a higher priority job executes less than its expected worst-case computation.*

This may happen because, even if $b \sqsupseteq a$, there may still exist values of v for which $c_a(v) < c_b(v)$. See for example the jobs in listings 8.1 and 8.2 and the computation functions in figs. 8.1a and 8.1b. A complete example is given below:

Example 8.3.2

A system consists of two jobs a and b, released simultaneously on 4 processors, with $d_a = 2.5$ and $d_b = 4$. $c_a(v)$ is given in fig. 8.3a. $c_b(v)$ is given in fig. 8.3b and will represent a WCET estimate of b. A computation function requiring less than worst-case computation, $c_{\text{less}}(v)$, is depicted in fig. 8.3c, and will represent the behavior of an actual execution of b.

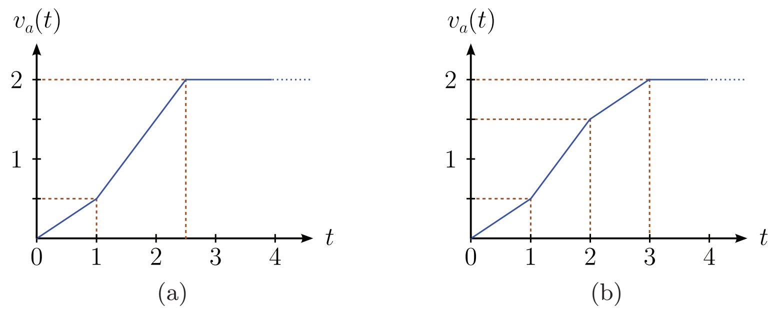


Figure 8.4: *Virtual times for Example 8.3.2*

Job b is given a higher priority than job a; this is necessary for b to meet its deadline given its worst-case computation. For b there are no higher priority jobs, and b always completes at or before its deadline of $t = 4$.

Job a suffers interference from b. If b executes its upper bound, then the virtual time of a progresses as shown in fig. 8.4a, and a completes at $t = 2.5$, meeting its deadline.

If b executes $c_{\text{less}}(t)$ instead, then the virtual time of a progresses as shown in fig. 8.4b. Job a suffers a more interference, and does not complete until $t = 3$, missing its deadline.

Observation 8.1 implies that exact schedulability analysis cannot be sustainable for the tasking model in this chapter: The system in example 8.3.2 is in fact schedulable when jobs execute their WCET, so an exact schedulability analysis must deem the system schedulable. On the other hand, the system is in fact not schedulable in some cases where jobs execute less, so a sustainable analysis cannot deem the system schedulable. It is also worth noting that observation 8.1 pertains to a wide range of job schedulers; nothing is assumed except for the existence of two jobs with a fixed relative priority.

8.4 Upper Bounds on Interference and Demand

A low priority job may suffer interference from higher priority jobs. In this section, an upper bound on interference will be derived as a function of the total demand of higher priority jobs. Then, an upper bound on the demand of a job will be given for windows that end with the deadline of the job. It will be shown that these upper bounds are also valid if jobs execute less than their upper bounds on computation.

8.4.1 Upper Bound on Interference

Definition 8.6 (Interference)
The interference to a job a in some window $[t_0, t]$ is denoted $\delta_a(t_0, t)$ and is defined as the difference between change in real and virtual time:

$$\delta_a(t_0, t) = (t - t_0) - (v_a(t) - v_a(t_0)) \tag{8.7}$$

Say a job a completes at time e. The components that contribute to the value of e are:

1. The release time r_a,
2. the length L_a of its reduced computation function,
3. the delay caused by interference from other tasks, $\delta(r_a, e)$, and
4. the skew from the intra job scheduler, which is at most $\sigma \cdot (e - r_a)$.

All of these components increase monotonically with e, so a sufficient condition for job a to meet its deadline is that $e \leq d_a$:

$$r_a + L_a + \delta_a(r_a, d_a) + \sigma \cdot (d_a - r_a) \leq d_a \tag{8.8}$$

Rearranging the terms yields

$$\delta_a(r_a, d_a) \leq (1 - \sigma) \cdot D_a - L_a \tag{8.9}$$

which is an upper bound to the amount of interference tolerated.

If one assumes that the computation function of a job is reduced to the number of CPUs, then interference is only caused by the execution of higher priority jobs. The amount of interference from these jobs depends on both their total amount of demand and on the structure of this demand. It also depends on the structure of the job being delayed. However, when using the fair intra-job scheduler there exists an upper bound on this interference that only depends on the total magnitude of demand from higher priority jobs:

Lemma 8.1. *Let x be a low priority job. Let m be the total number of processors, and $w(t)$ be the number of processors used by higher priority jobs at time t. Assume that x can never utilize more than m processors and let $\delta(t_0, t)$ be the total interference in the window $[t_0, t]$. Then*

$$\frac{\mathrm{d}}{\mathrm{d}t}\delta(t_0, t) \leq \frac{1}{m}w(t) \tag{8.10}$$

Proof. Differentiating the definition of interference (eq. (8.7)) with respect to the upper bound t yields

$$\frac{\mathrm{d}}{\mathrm{d}t}\delta(t_0, t) = 1 - \frac{\mathrm{d}}{\mathrm{d}t}v(t) \tag{8.11}$$

where $v(t)$ is the virtual time of x. Inserting the definition of virtual time (eq. (8.3)) yields

$$\frac{\mathrm{d}}{\mathrm{d}t}\delta(t_0, t) = 1 - \min\left\{\frac{s(t)}{c(v(t))}, 1\right\} \tag{8.12}$$

The number of processors available is the total number of processors minus the number of processors used by higher priority jobs:

$$s(t) = m - w(t) \tag{8.13}$$

Assume first that $w(t) > 0$ and that no processors are idle at time t. Then,

$$\frac{\mathrm{d}}{\mathrm{d}t}\delta(t_0, t) = 1 - \frac{m - w(t)}{c(v(t))} \tag{8.14}$$

Dividing by $w(t)$ results in

$$\frac{\mathrm{d}}{\mathrm{d}t}\delta(t_0, t) \cdot \frac{1}{w(t)} = \frac{c(v(t)) - m + w(t)}{c(v(t))\, w(t)} \tag{8.15}$$

In order to find the maximum value for the right-hand side with respect to $w(t)$, one can differentiate:

$$\frac{\mathrm{d}}{\mathrm{d}w(t)} \frac{c(v(t)) - m + w(t)}{c(v(t))\, w(t)} = \frac{m - c(v(t))}{c(v(t))\, w(t)^2} \tag{8.16}$$

The derivative is positive, unless $c(v(t)) = m$, in which case the derivative is zero. Inserting $w(t) = m$ (the maximum value of w) or $c(v(t)) = m$ into eq. (8.15), yields the same maximum right hand side of $1/m$. Therefore

$$\frac{\mathrm{d}}{\mathrm{d}t}\delta(t_0, t) \cdot \frac{1}{w(t)} \leq \frac{1}{m} \tag{8.17}$$

Reorganization then proves the lemma for $w(t) > 0$ and no idle processors. For the remaining cases one immediately gets $\frac{\mathrm{d}}{\mathrm{d}t}v(t) = 1$, and therefore, from eq. (8.11)

$$\frac{\mathrm{d}}{\mathrm{d}t}\delta(t_0, t) = 0 \tag{8.18}$$

which proves the lemma for $w(t) = 0$ or when processors are idle at t. □

Theorem 8.2. *Let $\delta(t_0, t_1)$ be the interference to a job caused by higher priority jobs in some window $[t_0, t_1]$. Let $w(t)$ be the number of processors used by higher priority jobs at time t. Let W be the total demand of the higher priority jobs in the window:*

$$W = \int_{t_0}^{t_1} w(t)\,\mathrm{d}t \tag{8.19}$$

Then

$$\delta(t_0, t_1) \le \frac{W}{m}. \tag{8.20}$$

Proof. Integrating both sides of eq. (8.10) over the interval results in the inequality

$$\int_{t_0}^{t_1} \frac{\mathrm{d}}{\mathrm{d}t}\delta(t_0, t)\,\mathrm{d}t \le \int_{t_0}^{t_1} \frac{1}{m} w(t)\,\mathrm{d}t \tag{8.21}$$

which can be simplified as follows

$$\begin{aligned} \int_{t_0}^{t_1} \frac{\mathrm{d}}{\mathrm{d}t}\delta(t_0, t)\,\mathrm{d}t &\le \frac{1}{m} \int_{t_0}^{t_1} w(t)\,\mathrm{d}t \\ \delta(t_0, t_1) - \delta(t_0, t_0) &\le \frac{1}{m} W \\ \delta(t_0, t_1) &\le \frac{W}{m} \end{aligned} \tag{8.22}$$

proving the theorem. □

8.4.2 Upper Bound on Demand

The bound in theorem 8.2 is useful for schedulability analysis, because it is independent of the structures of both the delayed and the interfering jobs, and only depends on the total amount of demand from the higher priority jobs. It is also sustainable with respect to decrease in computation, in the sense that any decrease in computation from the higher priority jobs will result in a reduction in worst-case delay.

In order to obtain a sustainable analysis for the whole system, a sustainable demand bound for higher priority jobs must also be found. Unfortunately, it is not feasible to derive tight and sustainable demand bounds for general windows in time based on the computation function of a job alone. This is illustrated by the following example:

Listings 8.4 to 8.6: *Jobs a, b and c in Example 8.4.1*

```
-- a
PAR
  SEQ
    Exec(1)
    PAR
      Exec(1)
      Exec(1)
      Exec(1)
  SEQ
    PAR
      Exec(1)
      Exec(1)
      Exec(1)
    Exec(1)
:
```

```
-- b
PAR
  Exec(1)
  Exec(1)
  Exec(1)
  SEQ
    PAR
      Exec(1)
      Exec(1)
      Exec(1)
    Exec(1)
:
```

```
-- c
PAR
  Exec(2)
  Exec(2)
  Exec(2)
  Exec(2)
:
```

Example 8.4.1
Consider the jobs a, b and c, depicted in listings 8.4 to 8.6, and the maximum demand of those jobs in the window $[0, 1]$. Job a is an upper bound of job b. Consider the demand in schedules where the jobs are given many processors, eg, $s(t) = 10$. The demands in $[0, 1]$ are

$$\mathcal{D}_{\mathrm{a}}(0, 1) = 4$$
$$\mathcal{D}_{\mathrm{b}}(0, 1) = 6$$

Here, the demand of the upper bound is lower. Now consider the job c. Its computation function is identical to $c_a(v)$, but for all $c' \sqsupseteq c$,

$$\mathcal{D}_{c'}(0, 1) \leq 4$$

so that removing computation from c will never yield a higher demand in $[0, 1]$ than c itself.

For more exaggerated job structures, the increase in demand caused by reduction in computation could be even higher. The computation function alone does not contain enough information to distinguish between jobs where executing less may lead to increase in demand, from jobs where it may not. However, this situation changes if one only considers windows where the ends coincide with the completion of the job.

Lemma 8.3. *If a is an upper bound of b, then for all v_0,*

$$\int_{v_0}^{L_b} c_b(v)\,\mathrm{d}v \leq \int_{v_0}^{L_a} c_a(v)\,\mathrm{d}v \tag{8.23}$$

Proof. First, note that $L_b \leq L_a$, and that $c_b(v) = 0$ when $v_b(t) > L_b$, so the upper limit of the right hand side integral can be set to L_a:

$$\int_{v_0}^{L_a} c_b(v)\,\mathrm{d}v \leq \int_{v_0}^{L_a} c_a(v)\,\mathrm{d}v \tag{8.24}$$

Beginning with the upper bound job a, say the value of an Exec statement is reduced by Δ, and that this part of the Exec statement contributed to $c_a(v)$ from virtual times v_x to $v_x + \Delta$. This removes Δ units of computation, but also means that the rest of the Exec statement, and computation in sequence to it, becomes ready to execute Δ virtual time units earlier. The computation in sequence is thus shifted towards lower values of v. Because the upper bounds of the integral coincides with the end of the job, there is no computation that can shift into the integral, and thus its value cannot increase.

By definition, when a is an upper bound of b, b can be obtained from a by reducing the values of Exec statements. Each such reduction will preserve the inequality, so for all b for which a is an upper bound, the lemma will hold. □

Theorem 8.4. *Let a be a job with deadline at d_a. Let $\mathcal{D}_{\text{ub}}(t_0, d_a)$ be an upper bound on the demand in the window $[t_0, d_a]$ for job a if a executes a computation function for which $c_a(v)$ is an upper bound. Then, for all schedules where a meets its deadline, and all $t_0 \leq d_a$,*

$$\mathcal{D}_{\text{ub}}(t_0, d_a) \leq \int_{L_a-(d_a-t_0)}^{L_a} c_a(v)\,\mathrm{d}v \tag{8.25}$$

Proof. The demand from a in a fixed window ending with a deadline of a obtains its maximum when a begins to execute as late as possible. If a meets its deadline, a may begin executing no later than $t = d_a - L_a$, in which case it must execute with $\frac{\mathrm{d}}{\mathrm{d}t}c_a(v) = 1$. This gives a virtual time for $t \leq d_a$ of

$$\begin{aligned} v_a(t) &= \max\{0,\ t_0 - (d_a - L_a)\} \\ &= \max\{0,\ L_a - (d_a - t_0)\} \end{aligned} \tag{8.26}$$

The definition of demand (eq. (8.6)) yields

$$\mathcal{D}_{\text{ub}}(t_0, d_a) \leq \int_{\max\{0,\ L_a-(d_a-t_0)\}}^{L_a} c_a(v)\,\mathrm{d}v \tag{8.27}$$

For negative values of v, $c_a(v) = 0$ by definition, so the max function can be removed:

$$\mathcal{D}_{\text{ub}}(t_0, d_a) \leq \int_{L_a-(d_a-t_0)}^{L_a} c_a(v)\,\mathrm{d}v \tag{8.28}$$

Finally, using lemma 8.3 it can be seen that this bound also holds even if a executes less than its upper bound $c_a(v)$. □

8.5 Schedulability Analysis for EDF

In this section, the framework from the previous sections will be used to demonstrate a sustainable schedulability analysis for systems using EDF.

The strategy is as follows: For each task, an upper bound on demand is found during any window ending with a deadline for the task, using theorem 8.4. This bound also holds when tasks execute less than their upper bound on computation.

With EDF, for a job a, only jobs with deadlines earlier than d_a may interfere with a. Using this property, an upper bound on the total demand from other tasks that may interfere with a is also found. Finally, theorem 8.2 is used to limit the interference caused by this demand. If the interference is sufficiently small then a will meet its deadline.

When analyzing tasks instead of jobs, it helps to extend computation functions so that they become periodic:

Definition 8.7
The T-period extension of a computation function $c(v)$ is denoted $c^\star(v, T)$ and is given by

$$c^\star(v, T) = c\left(v - \lfloor v/T \rfloor T\right) \tag{8.29}$$

Lemma 8.5. *The demand W from a task X in a window of length l from jobs with deadlines within the window, satisfies*

$$W \le \int_{L_X - l}^{L_X} c_X^\star(v, T_X)\,\mathrm{d}v \tag{8.30}$$

where $c_X(v)$ is an upper bound to the computation functions of all jobs of X.

Proof. Using a strategy from Bertogna and Cirinei [15], the jobs from task X that may execute within a window are divided into three categories: (1) jobs with release and deadline within the window, which contributes at most a full C_X of demand, (2) a maximum of one job with deadline outside the window, which by definition of W does not contribute to demand, and (3) a maximum of one job with release outside the window, which contributes up to C_X of demand, depending on the window alignment. An illustration is given in fig. 8.5a, with one job from each category.

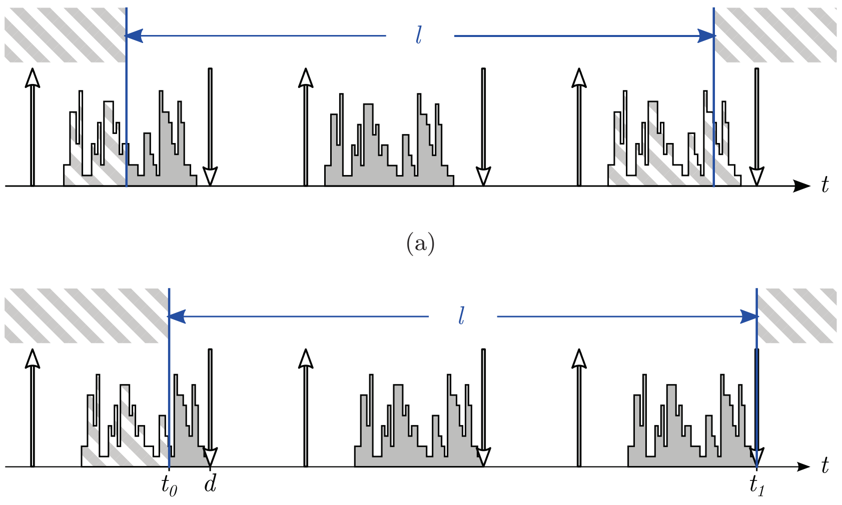


Figure 8.5: Alignment under EDF. (a) shows non-worst-case alignment, where last job in window contributes nothing to demand. (b) shows worst-case alignment.

A job with deadline outside of the window does not contribute to the demand, so the demand can never decrease by shifting the alignment of the task forward until a job deadline coincides with the end of the window. The demand from a job released outside the window is maximized by assuming that it executes as late as possible, completing execution at its deadline. Note that this assumption has no effect on the demand from jobs completely within the window. An upper bound on demand can therefore be found when the deadline of a job aligns with the end of the interval, with all earlier jobs released as late as possible. An illustration of the worst-case is given in fig. 8.5b.

A demand bound is then found by taking the sum of the contributions. Say the window is (t_0, t_1). Let the first job be denoted x, and the first job deadline d. The worst-case demand then satisfies

$$W \leq \mathcal{D}_x(t_0, d) + \left\lfloor \frac{l}{T_X} \right\rfloor C_X \tag{8.31}$$

Using theorem 8.4, one may write

$$W \leq \int_{L_X-(d-t_0)}^{L_X} c_X(v)\,\mathrm{d}v + \left\lfloor \frac{l}{T_X} \right\rfloor C_X \tag{8.32}$$

Replacing $c_X(v)$ with its T_X-period extension does not change the integral as long as $0 \leq v \leq T_x$:

$$W \leq \int_{L_X-(d-t_0)}^{L_X} c_X^\star(v, T_X)\,\mathrm{d}v + \left\lfloor \frac{l}{T_X} \right\rfloor C_X \tag{8.33}$$

Increasing the upper bound of the integral by T_X increases the value of the integral by C_X. Increasing the upper bound by $T_X \cdot \lfloor l/T_X \rfloor$ therefore cancels the second term entirely. As

$$T_X \cdot \left\lfloor \frac{l}{T_X} \right\rfloor = t_1 - d \tag{8.34}$$

this yields

$$W \leq \int_{L_x-(d-t_0)}^{L_x+(t_1-d)} c_x^\star(v, T_x)\,\mathrm{d}v \tag{8.35}$$

Because $c_x^\star$ is periodic with period T_x, if both limits of the integral is shifted by T_x this will not affect the value of the integral. As $t_1 - d$ is divisible by T_x, this results in

$$\begin{aligned} W &\leq \int_{L_X-(d-t_0)-(t_1-d)}^{L_X} c_X^\star(v, T_X)\,\mathrm{d}v \\ &\leq \int_{L_X+t_0-t_1}^{L_X} c_X^\star(v, T_X)\,\mathrm{d}v \end{aligned}$$

Finally, inserting $L = t_1 - t_0$ results in eq. (8.30). □

Theorem 8.6. *When using an* EDF *job scheduler, and the fair intra-job scheduler, a task A is schedulable if*

$$\frac{1}{m} \sum_{X \in \mathbb{T}:\ X \neq A} \int_{L_X-D_A}^{L_X} c_X^\star(v, T_X)\,\mathrm{d}v \ \leq\ D_A(1-\sigma) - L_A \tag{8.36}$$

where L_A is length of the longest reduced computation function of any job of A, $c_X(v)$ is an upper bound to the computation functions of all jobs of a task X and σ is the maximum skew caused by the intra-job scheduler. This schedulability test is sustainable with respect to a decrease in computation.

Proof. A single job of A executes for a window of D_A. Jobs from other tasks must have deadlines within this window to have higher priority than the job from A. Using lemma 8.5, the sum of demand from higher priority jobs of other tasks is found to satisfy

$$W_{\text{all}} \ \leq \sum_{X \in \mathbb{T}:\ X \neq A} \int_{L_X-D_A}^{L_X} c_X^\star(v, T_X)\,\mathrm{d}v \tag{8.37}$$

According to theorem 8.2 this causes a maximum interference of W_{all}/m. Inserting this into the condition for a job to meet its deadline, eq. (8.9), yields

$$\frac{1}{m} \sum_{X \in \mathbb{T}\colon X \neq A} \int_{L_X - D_A}^{L_X} c_X^{\star}(v, T_X)\,\mathrm{d}v \;\leq\; D_A(1-\sigma) - L_A \tag{8.38}$$

Both the upper bound on demand and the upper bound on interference hold when jobs execute less than their worst-case computations. The schedulability test is therefore sustainable. □

The above schedulability test has poor worst-case performance:

Observation 8.2. *It is possible to construct an implicit-deadline task set so that the schedulability test in theorem 8.6 has worst-case utilization arbitrarily close to 0.*

This will be demonstrated in the following example:

Example 8.5.1

A system contains two tasks, A and B, with relative deadlines equal to periods. Assume no skew ($\sigma = 0$). Let

$$T_B = \lambda \cdot T_A$$
$$c_A(v) = \begin{cases} 1 & \text{if } 0 \leq v \leq \epsilon \\ 0 & \text{otherwise} \end{cases}$$
$$c_B(v) = \begin{cases} m & \text{if } 0 \leq v \leq T_A \\ 0 & \text{otherwise} \end{cases}$$

for some $\lambda > 1$, $\epsilon > 0$. That is, there is a short task A that requires a small amount of computation on one CPU, and a longer task B that requires all CPUs for an interval identical to the period of A.

According to the test, A is schedulable if

$$\frac{1}{m} \int_{L_B - D_A}^{L_B} c_B^{\star}(v, T_B)\,\mathrm{d}v \leq D_A - L_x$$

which yields

$$\frac{1}{m} \int_{0}^{T_A} c_B^{\star}(v, T_B)\,\mathrm{d}v \leq T_A - \epsilon$$

Evaluating the integral results in $T_A \cdot m$. Simplifying yields

$$\epsilon \leq 0$$

Therefore, if $\epsilon > 0$ then A will not pass the schedulability test. The utilization of the system is

$$U = \frac{\epsilon}{T_A} + \frac{m T_A}{\lambda T_A} = \frac{\epsilon}{T_A} + \frac{m}{\lambda}$$

The utilization can be brought arbitrary close to zero by letting $\epsilon \approx 0$ and $\lambda \to \infty$.

8.6 Discussion

The EDF schedulability test of theorem 8.6 has a worst-case utilization arbitrarily close to 0. This bound is derived from one particular task set, and requires an infinitely long task period. It should be noted that this limit is caused by pessimism in the test; the task set used in the example is actually schedulable. The task model in this chapter is a superset of traditional multiprocessor EDF, which has an actual worst-case utilization arbitrarily close to 1.

The pessimism of the EDF test has several causes. Theorem 8.2 is pessimistic in that it assumes that demand from higher priority jobs align in the worst possible way. For example, if there is one low and one high priority job each requiring one processor, and $m \geq 2$, then demand from the high priority job will not cause interference. However, theorem 8.2 yields an interference bound of W/m. Theorem 8.4 is not pessimistic, as it is possible for a job to yield the demand given in the theorem. The main source of pessimism is the EDF test itself, which assumes that all higher priority jobs start as late as possible and finish at the same deadline. It should be possible to use theorems 8.2 and 8.4 to construct less pessimistic tests, perhaps by using iterated response-time analysis, as done in Bertogna and Cirinei [15].

The schedulability test is based on several assumptions, which to various degrees are sources of error: (1) Tasks are independent, (2) no cost of preemptions, (3) fair scheduler, with error bounded by skew variable, and (4) the system uses continuous time.

The first assumption severely limits the usefulness of the analysis for practical systems, but does not affect the validity of the analysis for systems where tasks actually are independent.

The second assumption is a source of error. Preemption causes jitter, and causes jobs to slow down because of cache misses etc. This should be taken into account. If an upper bound on the number of preemptions can be determined in advance, and an upper bound on the cost of each preemption can also be found, then the analysis can be modified to accommodate this cost.

The third assumption requires that a maximum value of skew can be found for the intra-job scheduler, so that worst-case skew grows linearly with time. This is an assumption that is not likely to hold for short windows of time, because it again would require an infinitely fast scheduler. There probably exist better models for skew. However, in this chapter total skew is only computed for windows from the release to the deadline for a job, which should of sufficient length to minimize the errors of the skew model itself.

The continuous time assumption is primarily required to prove the interference bound in theorem 8.2. It should be possible to incorporate the errors of this assumption as part of the errors caused by skew.

Another point to consider is whether to allow job-level parallelism at all. It is always possible to suppress JLP by serializing all jobs. If the resulting increase in response times is acceptable and does not cause deadlines to be missed, then this serialization can make task sets schedulable that are not schedulable when JLP is enabled. This is also discussed in Lakshmanan et al. [66], where job-level parallelism is suppressed for all jobs where it is not required for the job to meet its deadline.

Schedulability analysis of process-oriented systems on multiple processors is primarily limited by the lack of analysis techniques for synchronous communication: Whereas synchronous communication is required in all practical process-oriented systems, JLP is only required when there are parallel jobs that will miss deadlines when serialized. Still, real-time multiprocessor systems become increasingly common, and as the number of processing cores goes up, having to serialize each job will become an ever greater disadvantage. Moreover, when the typical problem of programming a multiprocessor system is to make computation more parallel, it is unfortunate if process-oriented systems, which are already parallel, must suppress this parallelism to be analyzable.

Part III

Closing Remarks

Chapter 9

Conclusions and Future Work

In computer science, a closure is a function together with a referencing environment for the non-local variables of that function

WIKIPEDIA

THE WORK on this thesis began as an interest in how timing requirements could be lifted to higher levels of abstraction, so to better integrate the computational and temporal aspects of programming real-time systems. Process-oriented programming was quickly selected as a suitable concurrency paradigm to use as basis, because it gives implementation of concurrency the high quality that it was hoped to achieve for implementation of temporal constraints. Later came the realization that process-oriented systems were poorly supported by schedulability analysis, which lead to the second part of the thesis.

9.1 Conclusions

In chapter 3, existing primitives for implementing temporal constraints were evaluated with respect to program quality. It was found that in general, using primitives at a low level of abstraction leads to low quality programs. The use of primitives that explicitly set scheduling priorities was strongly discouraged. Moreover, it was argued that a real-time language targeting reactive systems should have primitives that explicitly references relative times, as converting relative times from a specification into absolute times incurs a complexity which obfuscates the programmer's intention and reduces readability.

The time-construct was then introduced, which sets an equal minimum and maximum response time constraint for a block of code, and it was demonstrated that this construct can be used to implement a wide range of temporal constraints in a readable and intuitive manner.

One conclusion was that a process subject to a deadline constraint should not communicate synchronously with a process that may be subject to a delay constraint. Raising events and triggering sporadic tasks fall under the former category, while event handlers and the tasks themselves fall under the latter. Triggering mechanisms should therefore not be synchronous.

A new programming language, Toc, was designed in chapter 4, and a compiler and run-time system were implemented. Toc is based on occam, but replaces all occam primitives that handle timing with the primitives suggested in chapter 3. Toc also has a radically different execution model, and uses an integrated earliest deadline first (EDF) scheduler, rather than the non-real-time scheduling model employed by occam.

When programming a real-time system one may allow background tasks, ie, processes that are never considered urgent, and which therefore should yield to any task with a specified deadline. The alternative is not to allow these tasks, so that all functionality of the system must be under the responsibility of a deadline constraint in order to be executed. This is the stance taken by the Toc scheduler. This strategy, called lazy scheduling, forces the programmer to become aware of all the timing requirements present in a system, as code with no requirement will not be executed, even if the system is not doing anything else. The upside of this is that it becomes easier to spot missing or incorrectly assigned deadlines, errors that would otherwise only be revealed when the system load is high. The downside is that enforcing lazy scheduling may require the introduction of artificial or obscure deadlines.

Part II discussed schedulability analysis of process-oriented systems. These have two properties that are incompatible with traditional schedulability analysis, (1) they use synchronous communication, and (2) they have complex parallel structures. The latter is only relevant for multiprocessor systems.

A new method for analyzing schedulability of systems using synchronous communication was developed in chapter 6. The analysis was based on organizing communication according to a client-server pattern, while preventing real-time tasks from acting as servers. Some restrictions on communication were added in order to guarantee the absence of deadlocks; the resulting analysis was not much more complicated than existing equivalents for shared memory based systems.

Two application examples were shown and discussed. In one example, the analysis was applied to a system using the priority inheritance protocol (PIP) and mutual exclusion synchronization in the form of protected objects, and it was demonstrated that the new analysis may yield less pessimistic bounds than the traditional analysis technique. In the other example it was shown that the schedulability of a system can sometimes be improved by deferring computation generated by a server call until after the call, an optimization which is difficult to achieve in a system that uses mutual exclusion based communication.

Chapter 7 described a mathematical model for reasoning on programs with complex parallel structures. The model can be used as a tool for analyzing non-communicating real-time jobs. Several instances of counter-intuitive behavior were demonstrated to be inherent in real-time systems where processes are allowed such a complex structure. For example, a process may be guaranteed to complete on schedule, if and only if computation is not removed from the process.

Chapter 8 introduced a framework for analyzing complete systems of such processes, if the systems use an intra-job scheduler that is reasonably fair. The notion of upper bounds on computation was discussed for these systems, and it was demonstrated that even under the fair scheduler, jobs executing the upper bound does not necessarily represent a worst-case scenario.

An upper bound of interference as a function of demand from higher priority jobs was derived, as well as an upper bound on demand from a job for windows that end with a deadline for the job. These bounds were used to construct an EDF schedulability test. This test is fairly simple, and has poor worst-case performance, but allows analysis for a class of systems for which no existing analysis methods currently apply. Moreover, it is believed that the interference and demand bounds can be used to create schedulability tests with better performance.

9.2 Future Work

PhD work is also subject to temporal constraints. With no attempt at making a complete list, here are some possible topics for future work:

1. To try the time-construct in a language that uses entry-type synchronous communication between processes. This structure is convenient both for schedulability analysis and for guaranteeing the absence of deadlock, and would also solve the dilemma of the default extended rendezvous described in section 4.4.6.

2. To explore the use of light-weight processes in real-time systems, and the balance between responsiveness required by real-time constraints, and the low overhead that can be achieved when using non-preemptive scheduling.
3. To improve the Toc compiler in various ways, such as implementing an analytical usage checker to replace the existing one (which is based on simulation), and to add warnings for code that will never be executed due to excessive laziness.
4. To analyze schedulability of process-oriented client–server systems on multiprocessors. This will require new synchronization protocols, as existing multiprocessor protocols assume short critical sections, which are not realistic in process-oriented systems.
5. To identify necessary or sufficient conditions for when deferring computation in a server call may improve the schedulability of a system.
6. To extend the computation time process (CTP) model to handle communication, and to handle multiple jobs, so that schedulability analysis can be performed that is independent of the intra-job scheduler.
7. To investigate further the notion of safe upper bounds on CTPs, perhaps to find out whether there exists a unique, best safe upper bound for every CTP, or to develop an algorithm that finds these bounds.
8. To improve the EDF schedulability test for systems of malleable jobs when using the fair intra-job scheduler, eg, by using iteration.

Concluding Remarks

Process-oriented systems lend themselves more easily to formal verification, scale better, and are arguably easier to write correctly. It is therefore unfortunate that the paradigm is not widely used in real-time systems. The reasons for this include the lack of suitable primitives for implementing real-time constraints, as well as the lack of well-known schedulability analysis techniques. The results from this thesis may help to form a basis for more widespread use of process-oriented programming in real-time systems.

Part IV

Appendices

Appendix A

Toc Language Specification in BNF

THIS CHAPTER describes the syntax and lexical structure of Toc. Parts of this text is auto-generated by `bnfc` [81] based on a Backus–Naur form (BNF) language description that is also used to generate the code for the parser.

The Lexical Structure of Toc

This section describes the literals, keywords, symbols and comments available in Toc.

Literals

Integer literals ⟨*Int*⟩ are nonempty sequences of digits. Double-precision float literals ⟨*Double*⟩ have the structure indicated by the regular expression

⟨*digit*⟩+ `'.'` ⟨*digit*⟩+ ((`'e'` | `'E'`) `'-'`? ⟨*digit*⟩+)?

that is, two sequences of digits separated by a decimal point, optionally followed by an unsigned or negative exponent. String literals ⟨*String*⟩ have the form `"`*x*`"`, where *x* is any sequence of any characters except `"` unless preceded by \. Character literals ⟨*Char*⟩ have the form `'`*c*`'`, where *c* is any single character.

Identifiers come in three classes: upper-case identifiers, lower-case identifiers and capitalized identifiers. Capitalized identifiers are used for procedures to simplify parsing; that they are required should be considered as a limitation of the compiler. The ⟨*UpperIdent*⟩, ⟨*LowerIdent*⟩ and ⟨*CapIdent*⟩ identifier literals are recognized by the following regular expressions, respec-

Table A.1: *Reserved Keywords in Toc*

ALT	AND	BITAND	BITOR	BOOL
BYTESIN	CASE	CHAN	CLEAR	DATA
DAY	ELSE	EVENT	EXTERN	FALSE
FROM	FUNCTION	HANDLE	HOUR	IF
INT	IS	MIN	MSEC	NOT
NSEC	OR	PAR	PRINT	PROC
PROTOCOL	RAISE	REAL	RECORD	REM
SEC	SEQ	SIZE	SKIP	STOP
TIME	TIMEOUT	TIMESPEC	TRUE	USEC
VAL	VALOF	WHILE		

Table A.2: *Symbols in Toc*

:=	(	)	=	&	?	!	;	,
:	[	]	<>	<	<	<=	>=	><
<<	>>	+	−	*	/	~		

tively:

$$\begin{aligned} \langle upper\rangle\ (\langle upper\rangle \mid \langle lower\rangle \mid \langle digit\rangle \mid \text{'\_'} \mid \text{'.'})* \\ \langle lower\rangle\ (\langle lower\rangle \mid \langle lower\rangle \mid \langle digit\rangle \mid \text{'\_'} \mid \text{'.'})* \\ \langle upper\rangle\ \langle lower\rangle\ (\langle upper\rangle \mid \langle lower\rangle \mid \langle digit\rangle \mid \text{'\_'} \mid \text{'.'})* \end{aligned}$$

Reserved Words and Symbols

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers (eg, they are allowed to be adjacent). The lexer follows the usual rules for lexing, including longest match and spacing conventions. The reserved keywords and symbols of Toc are listed in tables A.1 and A.2, respectively.

Comments

Toc allows single-line comments, which begin with --.

Layout Resolution

After lexing, but before parsing, the symbol list undergoes layout resolution. A line-break after `,` (comma), FOR, FROM, or IS is ignored; otherwise, after each line-break, the indentation of the next symbol is compared to the indentation of the first symbol on the previous line. If the indentation is greater, a `{`-token is inserted. If the indentation is equal, then a `$` is inserted. If the indentation is less, then a suitable number of `}` are inserted, to close all opening braces since indentation exceeded the level of the new line.

The Syntactic Structure of Toc in Bnf

This section lists the syntax of Toc in BNF. Non-terminals are enclosed between ⟨ and ⟩. The symbols ::= (production), | (union) and ϵ (empty rule) belong to the BNF notation. The symbols `{`, `$` and `}` are tokens inserted during layout resolution. All other symbols are terminals.

⟨*AnyIdent*⟩	::=	⟨*LowerIdent*⟩
	|	⟨*UpperIdent*⟩
	|	⟨*CapIdent*⟩
⟨*Program*⟩	::=	⟨*ListDecl*⟩
⟨*Block*⟩	::=	`{`⟨*ListStmt*⟩`}`
⟨*Stmt*⟩	::=	⟨*ListTermDecl*⟩⟨*Proc*⟩
⟨*ListStmt*⟩	::=	⟨*Stmt*⟩
	|	⟨*Stmt*⟩ `$` ⟨*ListStmt*⟩

```
⟨Proc⟩            ::=  ⟨Alt⟩
                   |   ⟨ListNonEmptyExp⟩ := ⟨ListNonEmptyExp⟩
                   |   ⟨CapIdent⟩ ( ⟨ListExpr⟩ )
                   |   ⟨Comm⟩
                   |   RAISE ⟨Expr⟩
                   |   CLEAR ⟨Expr⟩
                   |   ⟨Handle⟩
                   |   ⟨CaseBlock⟩
                   |   ⟨IfChoice⟩
                   |   PAR ⟨LowerIdent⟩ =  ⟨Expr⟩ FOR ⟨Expr⟩ ⟨Block⟩
                   |   SEQ ⟨LowerIdent⟩ =  ⟨Expr⟩ FOR ⟨Expr⟩ ⟨Block⟩
                   |   SEQ ⟨Block⟩
                   |   PAR ⟨Block⟩
                   |   PRINT ⟨ListExpr⟩
                   |   SKIP
                   |   STOP
                   |   TIME ⟨Expr⟩ ⟨Block⟩
                   |   WHILE ⟨Expr⟩ ⟨Block⟩
⟨Alt⟩             ::=  ALT { ⟨ListAltStmt⟩ }
                   |   ALT ⟨LowerIdent⟩ =  ⟨Expr⟩ FOR ⟨Expr⟩ {⟨AltStmt⟩ }
⟨AltStmt⟩         ::=  ⟨ListTermDecl⟩ ⟨Alt⟩
                   |   ⟨ListTermDecl⟩ ⟨Guard⟩
⟨ListAltStmt⟩     ::=  ⟨AltStmt⟩
                   |   ⟨AltStmt⟩ $ ⟨ListAltStmt⟩
⟨Guard⟩           ::=  ⟨Expr⟩ & ⟨Comm⟩
                   |   ⟨Comm⟩
                   |   ⟨Expr⟩ & SKIP ⟨Block⟩
⟨Input⟩           ::=  ?
                   |   ??
⟨Output⟩          ::=  !
                   |   !!
⟨Comm⟩            ::=  ⟨Expr⟩ ⟨Output⟩ ⟨ListCommExpr⟩
                   |   ⟨Expr⟩ ⟨Output⟩ ⟨ListCommExpr⟩ ⟨Block⟩
                   |   ⟨Expr⟩ ⟨Input⟩ ⟨ListCommExpr⟩
                   |   ⟨Expr⟩ ⟨Input⟩ ⟨ListCommExpr⟩ ⟨Block⟩
                   |   ⟨Expr⟩ ⟨Input⟩ CASE { ⟨ListCommCaseExpr⟩ }
⟨CommExpr⟩        ::=  ⟨Expr⟩

⟨ListCommExpr⟩
                  ::=  ⟨CommExpr⟩
                   |   ⟨CommExpr⟩ ; ⟨ListCommExpr⟩
⟨CommCaseExpr⟩
                  ::=  ⟨ListCommExpr⟩ ⟨Block⟩
```

$\langle$*ListCommCaseExpr*$\rangle$		
	::=	$\langle$*CommCaseExpr*$\rangle$
	|	$\langle$*CommCaseExpr*$\rangle$ `$` $\langle$*ListCommCaseExpr*$\rangle$
$\langle$*Handle*$\rangle$	::=	`HANDLE {` $\langle$*HandleStmt*$\rangle\langle$*TimeoutStmt*$\rangle$ `}`
	|	`HANDLE` $\langle$*HandleStmt*$\rangle$
$\langle$*HandleStmt*$\rangle$	::=	$\langle$*Expr*$\rangle\langle$*Block*$\rangle$
$\langle$*TimeoutStmt*$\rangle$	::=	`$ TIMEOUT` $\langle$*Expr*$\rangle\langle$*Block*$\rangle$
	|	ϵ
$\langle$*CaseBlock*$\rangle$	::=	`CASE` $\langle$*Expr*$\rangle$ `{` $\langle$*ListCaseAlt*$\rangle$ `}`
$\langle$*CaseAlt*$\rangle$	::=	$\langle$*ListNonEmptyExp*$\rangle\langle$*Block*$\rangle$
	|	`ELSE` $\langle$*Block*$\rangle$
$\langle$*ListCaseAlt*$\rangle$	::=	$\langle$*CaseAlt*$\rangle$
	|	$\langle$*CaseAlt*$\rangle$ `$` $\langle$*ListCaseAlt*$\rangle$
$\langle$*NonEmptyExp*$\rangle$		
	::=	$\langle$*Expr*$\rangle$
$\langle$*ListNonEmptyExp*$\rangle$		
	::=	$\langle$*NonEmptyExp*$\rangle$
	|	$\langle$*NonEmptyExp*$\rangle$ `,` $\langle$*ListNonEmptyExp*$\rangle$
$\langle$*IfChoice*$\rangle$	::=	`IF {` $\langle$*ListIfStmt*$\rangle$ `}`
	|	`IF` $\langle$*LowerIdent*$\rangle$ `=` $\langle$*Expr*$\rangle$ `FOR` $\langle$*Expr*$\rangle$ `{` $\langle$*IfStmt*$\rangle$ `}`
$\langle$*IfStmt*$\rangle$	::=	$\langle$*IfChoice*$\rangle$
	|	$\langle$*Expr*$\rangle\langle$*Block*$\rangle$
$\langle$*ListIfStmt*$\rangle$	::=	$\langle$*IfStmt*$\rangle$
	|	$\langle$*IfStmt*$\rangle$ `$` $\langle$*ListIfStmt*$\rangle$
$\langle$*TimeUnit*$\rangle$	::=	`DAY`
	|	`HOUR`
	|	`MIN`
	|	`SEC`
	|	`MSEC`
	|	`USEC`
	|	`NSEC`
$\langle$*ValOf*$\rangle$	::=	$\langle$*ListTermDecl*$\rangle$ `VALOF`
		`{` $\langle$*Stmt*$\rangle$ `$ RESULT` $\langle$*ListNonEmptyExp*$\rangle$ `}`
$\langle$*ProtocolDecl*$\rangle$	::=	`PROTOCOL` $\langle$*AnyIdent*$\rangle\langle$*ProtoDef*$\rangle$
$\langle$*ProtoDef*$\rangle$	::=	`IS` $\langle$*ListSeqType*$\rangle$
	|	`{ CASE {` $\langle$*ListProtoCase*$\rangle$ `} } $`
$\langle$*SeqType*$\rangle$	::=	$\langle$*VarType*$\rangle$

⟨*ListSeqType*⟩	::=	⟨*SeqType*⟩
	|	⟨*SeqType*⟩ `;` ⟨*ListSeqType*⟩
⟨*ProtoCase*⟩	::=	⟨*LowerIdent*⟩
	|	⟨*LowerIdent*⟩ `;` ⟨*ListSeqType*⟩
⟨*ListProtoCase*⟩	::=	⟨*ProtoCase*⟩
	|	⟨*ProtoCase*⟩ `$` ⟨*ListProtoCase*⟩
⟨*DataType*⟩	::=	`DATA TYPE` ⟨*AnyIdent*⟩ ⟨*DataDef*⟩
⟨*DataDef*⟩	::=	`IS` ⟨*VarType*⟩
	|	`{ RECORD {` ⟨*ListDecl*⟩ `} } $`
⟨*Function*⟩	::=	⟨*ListVarType*⟩ `FUNCTION` ⟨*AnyIdent*⟩ `(` ⟨*ListParam*⟩ `)` ⟨*FuncDef*⟩
⟨*EFunction*⟩	::=	⟨*VarType*⟩ `FUNCTION` ⟨*AnyIdent*⟩ `(` ⟨*ListParam*⟩ `)`
⟨*FuncDef*⟩	::=	`{` ⟨*ValOf*⟩ `} $`
	|	`IS` ⟨*ListNonEmptyExp*⟩
⟨*Procedure*⟩	::=	`PROC` ⟨*AnyIdent*⟩ `(` ⟨*ListParam*⟩ `)` ⟨*Block*⟩ `$`
⟨*Param*⟩	::=	⟨*VarType*⟩ ⟨*DeclId*⟩
	|	⟨*DeclId*⟩
⟨*ListParam*⟩	::=	ϵ
	|	⟨*Param*⟩
	|	⟨*Param*⟩ `,` ⟨*ListParam*⟩
⟨*Decl*⟩	::=	⟨*VarType*⟩ ⟨*ListDeclId*⟩ `:`
	|	⟨*DataType*⟩ `:`
	|	⟨*ProtocolDecl*⟩ `:`
	|	⟨*Function*⟩ `:`
	|	`EXTERN` ⟨*EFunction*⟩ `:`
	|	⟨*Procedure*⟩ `:`
⟨*ListDecl*⟩	::=	ϵ
	|	⟨*Decl*⟩
	|	⟨*Decl*⟩ `$` ⟨*ListDecl*⟩
⟨*TermDecl*⟩	::=	⟨*Decl*⟩
⟨*ListTermDecl*⟩	::=	ϵ
	|	⟨*TermDecl*⟩ `$` ⟨*ListTermDecl*⟩
⟨*Qualifier*⟩	::=	`VAL`
	|	ϵ
⟨*Channel*⟩	::=	ϵ
	|	`CHAN` ⟨*ListArrayStmt*⟩
⟨*VarType*⟩	::=	⟨*Qualifier*⟩ ⟨*Channel*⟩ ⟨*Primitive*⟩ ⟨*ListArrayStmt*⟩

⟨*ListVarType*⟩	::=	⟨*VarType*⟩
	|	⟨*VarType*⟩ , ⟨*ListVarType*⟩
⟨*Primitive*⟩	::=	`BYTE`
	|	`BOOL`
	|	`INT`
	|	`REAL`
	|	`TIMESPEC`
	|	`EVENT`
	|	⟨*UpperIdent*⟩
⟨*ArrayStmt*⟩	::=	`[` ⟨*Expr*⟩ `]`
	|	`[ ]`
⟨*ListArrayStmt*⟩	::=	ϵ
	|	⟨*ArrayStmt*⟩⟨*ListArrayStmt*⟩
⟨*DeclId*⟩	::=	⟨*LowerIdent*⟩⟨*ChanDir*⟩⟨*Abbrev*⟩
⟨*ListDeclId*⟩	::=	⟨*DeclId*⟩
	|	⟨*DeclId*⟩ , ⟨*ListDeclId*⟩
⟨*ChanDir*⟩	::=	ϵ
	|	`?`
	|	`!`
⟨*Abbrev*⟩	::=	ϵ
	|	`IS` ⟨*Expr*⟩
⟨*PosConst*⟩	::=	⟨*TokInteger*⟩
	|	⟨*TokDouble*⟩
	|	⟨*TokString*⟩
	|	`TRUE`
	|	`FALSE`
	|	⟨*TokChar*⟩
⟨*Const*⟩	::=	⟨*Integer*⟩
	|	⟨*Double*⟩
	|	⟨*String*⟩
	|	⟨*Char*⟩
	|	⟨*ListConst*⟩
⟨*ListConst*⟩	::=	ϵ
	|	⟨*Const*⟩
	|	⟨*Const*⟩ , ⟨*ListConst*⟩
⟨*IntConst*⟩	::=	⟨*Integer*⟩⟨*Integer*⟩⟨*Const*⟩
⟨*Expr2*⟩	::=	⟨*Expr2*⟩ `OR` ⟨*Expr3*⟩
	|	⟨*Expr3*⟩
⟨*Expr3*⟩	::=	⟨*Expr3*⟩ `AND` ⟨*Expr4*⟩
	|	⟨*Expr4*⟩

⟨*Expr4*⟩	::=	⟨*Expr4*⟩ `=` ⟨*Expr5*⟩
	|	⟨*Expr4*⟩ `<>` ⟨*Expr5*⟩
	|	⟨*Expr5*⟩
⟨*Expr5*⟩	::=	⟨*Expr5*⟩ `<` ⟨*Expr6*⟩
	|	⟨*Expr5*⟩ `>` ⟨*Expr6*⟩
	|	⟨*Expr5*⟩ `<=` ⟨*Expr6*⟩
	|	⟨*Expr5*⟩ `>=` ⟨*Expr6*⟩
	|	⟨*Expr6*⟩
⟨*Expr6*⟩	::=	⟨*Expr6*⟩ `BITOR` ⟨*Expr7*⟩
	|	⟨*Expr7*⟩
⟨*Expr7*⟩	::=	⟨*Expr7*⟩ `><` ⟨*Expr8*⟩
	|	⟨*Expr8*⟩
⟨*Expr8*⟩	::=	⟨*Expr8*⟩ `BITAND` ⟨*Expr9*⟩
	|	⟨*Expr9*⟩
⟨*Expr9*⟩	::=	⟨*Expr9*⟩ `<<` ⟨*Expr10*⟩
	|	⟨*Expr9*⟩ `>>` ⟨*Expr10*⟩
	|	⟨*Expr10*⟩
⟨*Expr10*⟩	::=	⟨*Expr10*⟩ `+` ⟨*Expr11*⟩
	|	⟨*Expr10*⟩ `-` ⟨*Expr11*⟩
	|	⟨*Expr11*⟩
⟨*Expr11*⟩	::=	⟨*Expr11*⟩ ⟨*TimeUnit*⟩
	|	⟨*Expr11*⟩ `*` ⟨*Expr12*⟩
	|	⟨*Expr11*⟩ `/` ⟨*Expr12*⟩
	|	⟨*Expr11*⟩ `REM` ⟨*Expr12*⟩
	|	⟨*Expr12*⟩
⟨*Expr13*⟩	::=	`-` ⟨*Expr12*⟩
	|	`+` ⟨*Expr12*⟩
	|	`NOT` ⟨*Expr12*⟩
	|	`~` ⟨*Expr12*⟩
	|	`SIZE` ⟨*Expr13*⟩
	|	`BYTESIN` ⟨*Expr13*⟩
	|	`BYTESIN` ⟨*VarType*⟩
	|	⟨*Expr14*⟩
⟨*Expr14*⟩	::=	⟨*LowerIdent*⟩ `(` ⟨*ListExpr*⟩ `)`
	|	⟨*Expr14*⟩ `[` ⟨*Expr*⟩ `]`
	|	⟨*Expr15*⟩
⟨*Expr15*⟩	::=	⟨*PosConst*⟩
	|	`NOW`
	|	⟨*LowerIdent*⟩
	|	⟨*ExprList*⟩
	|	`(` ⟨*Expr*⟩ `)`
⟨*Expr*⟩	::=	⟨*Expr1*⟩

⟨*Expr1*⟩	::=	⟨*Expr2*⟩
⟨*Expr12*⟩	::=	⟨*Expr13*⟩
⟨*ListExpr*⟩	::=	ϵ
	|	⟨*Expr*⟩
	|	⟨*Expr*⟩ `,` ⟨*ListExpr*⟩
⟨*ExprList*⟩	::=	`[` ⟨*ListExpr*⟩ `]`
	|	`[` ⟨*Expr*⟩ `FROM` ⟨*Expr*⟩ `]`
	|	`[` ⟨*Expr*⟩ `FROM` ⟨*Expr*⟩ `FOR` ⟨*Expr*⟩ `]`
	|	`[` ⟨*Expr*⟩ `FOR` ⟨*Expr*⟩ `]`

Appendix B

Complete Listing of Elevator Control Code

THIS SECTION lists the complete code for the elevator example discussed in section 4.6.

Listing B.1: *Elevator Example: Complete Listing*

```
-- *** CONSTANTS AND DATA TYPES ***--{{{
DATA TYPE FLOOR IS INT:
DATA TYPE DIR IS INT:
VAL INT floors IS 4:
VAL INT dirs IS 3:
VAL DIR dir.up IS 0:
VAL DIR dir.down IS 1:
VAL DIR dir.none IS 2:

PROTOCOL NULL
  CASE
    null
:
--}}}
-- *** ELEVATOR INTERFACE ***--{{{
PROTOCOL INTERFACE.BUTTON
  CASE
    call ; FLOOR ; DIR
    goto ; FLOOR
:

PROTOCOL INTERFACE.BUTTON.LIGHT
  CASE
    call ; FLOOR ; DIR ; BOOL
    goto ; FLOOR ; BOOL
:

-- *** IO MODULE ***--{{{
PROTOCOL IO.INPUT
  CASE
    call ; FLOOR ; DIR
    goto ; FLOOR
    sensor ; FLOOR
    obstruction ; BOOL
    panic.button
```

```
:

PROTOCOL IO.OUTPUT
  CASE
    goto.light ; FLOOR ; BOOL
    call.light ; FLOOR ; DIR ; BOOL
    floor.indicator ; FLOOR
    door.light ; BOOL
    panic.light ; BOOL
    motor.speed ; INT
:

EXTERN INT FUNCTION comedi_open(VAL BYTE[] filename):
EXTERN INT FUNCTION comedi_dio_config(VAL INT handle, VAL INT subdevice, VAL INT channel,
    VAL INT direction):
EXTERN INT FUNCTION comedi_dio_write(VAL INT handle, VAL INT subdevice, VAL INT channel,
    VAL BOOL bit):
EXTERN INT FUNCTION comedi_dio_read(VAL INT handle, VAL INT subdevice, VAL INT channel,
    BOOL bit):
EXTERN INT FUNCTION comedi_data_write(VAL INT handle, VAL INT subdevice, VAL INT channel,
    VAL INT range, VAL INT ref, VAL INT data):
EXTERN INT FUNCTION comedi_data_read(VAL INT handle, VAL INT subdevice, VAL INT channel,
    VAL INT range, VAL INT ref, INT data):
EXTERN INT FUNCTION abs(VAL INT x):

PROC Io.Set.Bit(VAL INT handle, channel, VAL BOOL value)
  INT dummy:
  dummy := comedi_dio_write(handle, channel >> 8, channel BITAND 255, value)
:
PROC Io.Read.Bit(VAL INT handle, channel, BOOL bit)
  INT dummy:
  dummy := comedi_dio_read(handle, channel >> 8, channel BITAND 255, bit)
:
PROC Io.Write.Analog(VAL INT handle, VAL INT channel, VAL INT value)
  INT dummy:
  VAL INT ground IS 0:
  dummy := comedi_data_write(handle, channel >> 8, channel BITAND 255, 0, ground, value)
:

PROC Io.Read.Analog(VAL INT handle, VAL INT channel, INT value)
  INT dummy:
  VAL INT ground IS 0:
  SEQ
    value := 0
    dummy := comedi_data_write(handle, channel >> 8, channel BITAND 255, 0, ground, value)
:

-- *** IO PORTS *** --{{{
VAL INT port.port4 IS 3:
VAL INT port.obstruction IS (768+23):
VAL INT port.panic IS (768+22):
VAL INT port.goto1 IS (768+21):
VAL INT port.goto2 IS (768+20):
VAL INT port.goto3 IS (768+19):
VAL INT port.goto4 IS (768+18):
VAL INT port.call.up1 IS (768+17):
VAL INT port.call.up2 IS (768+16):
```

```
VAL INT port.port1 IS 2:
VAL INT port.call.down2 IS (512+0):
VAL INT port.call.up3 IS (512+1):
VAL INT port.call.down3 IS (512+2):
VAL INT port.call.down4 IS (512+3):
VAL INT port.sensor1 IS (512+4):
VAL INT port.sensor2 IS (512+5):
VAL INT port.sensor3 IS (512+6):
VAL INT port.sensor4 IS (512+7):

VAL INT port.port3 IS 3:
VAL INT port.motordir IS (768+15):
VAL INT port.panic.light IS (768+14):
VAL INT port.light.goto1 IS (768+13):
VAL INT port.light.goto2 IS (768+12):
VAL INT port.light.goto3 IS (768+11):
VAL INT port.light.goto4 IS (768+10):
VAL INT port.light.up1 IS (768+9):
VAL INT port.light.up2 IS (768+8):

VAL INT port.port2 IS 3:
VAL INT port.light.down2 IS (768+7):
VAL INT port.light.up3 IS (768+6):
VAL INT port.light.down3 IS (768+5):
VAL INT port.light.down4 IS (768+4):
VAL INT port.door.open IS (768+3):
VAL INT port.floor.ind2 IS (768+1):
VAL INT port.floor.ind1 IS (768+0):

VAL INT port.port0 IS 1:
VAL INT port.motor IS (256+0):

--}}}
PROC Io.Init(INT handle)
  INT status:
  SEQ
    handle := comedi_open("/dev/comedi0")
    VAL INT comedi.input IS 0:
    VAL INT comedi.output IS 1:
    IF
      handle = 0
        SKIP
      TRUE
        SEQ i = 0 FOR 8
          SEQ
            status := status BITOR comedi_dio_config(handle, port.port1, i, comedi.input)
            status := status BITOR comedi_dio_config(handle, port.port2, i, comedi.output)
            status := status BITOR comedi_dio_config(handle, port.port3, i+8, comedi.output)
            status := status BITOR comedi_dio_config(handle, port.port4, i+16, comedi.input)
    IF
      handle = 0 OR status = -1
        SEQ
          PRINT "unable to open initialize hardware"
          STOP
      TRUE
        PRINT "hardware initialized"
```

```
:

PROC Io.Output(VAL INT handle, CHAN IO.OUTPUT output)
  VAL INT [floors] port.light.goto IS [port.light.goto1, port.light.goto2, port.light.goto3, port.light.goto4]:
  VAL INT [floors] port.light.down IS [−1, port.light.down2, port.light.down3, port.light.down4]:
  VAL INT [floors] port.light.up IS [port.light.up1, port.light.up2, port.light.up3, −1]:
  BOOL previous.motor.dir:
  WHILE TRUE
    INT floor:
    BOOL setting:
    INT speed:
    DIR dir:
    output ?? CASE
      goto.light ; floor ; setting
        Io.Set.Bit(handle, port.light.goto[floor], setting)
      call.light ; floor ; dir ; setting
        IF
          dir = dir.down
            Io.Set.Bit(handle, port.light.down[floor], setting)
          TRUE
            Io.Set.Bit(handle, port.light.up[floor], setting)
      floor.indicator ; floor
        SEQ
          Io.Set.Bit(handle, port.floor.ind1, floor / 2 = 1)
          Io.Set.Bit(handle, port.floor.ind2, floor REM 2 = 1)
      door.light ; setting
        Io.Set.Bit(handle, port.door.open, setting)
      panic.light ; setting
        Io.Set.Bit(handle, port.panic.light, setting)
      motor.speed ; speed
        IF
          speed = 0
            SEQ
              Io.Set.Bit(handle, port.motordir, NOT previous.motor.dir)
              TIME 5 MSEC −− breaking duration
                SKIP
              Io.Write.Analog(handle, port.motor, 0)
          TRUE
            SEQ
              previous.motor.dir := speed < 0
              Io.Set.Bit(handle, port.motordir, previous.motor.dir)
              Io.Write.Analog(handle, port.motor, 2048+abs(speed))
:

PROC Io.Input(VAL INT handle, CHAN IO.INPUT input)
  BOOL [floors] call.down.state:
  BOOL [floors] call.up.state:
  BOOL [floors] goto.state:
  BOOL [floors] sensor.state:
  VAL INT [floors] port.call.down IS [−1, port.call.down2, port.call.down3, port.call.down4]:
  VAL INT [floors] port.call.up IS [port.call.up1, port.call.up2, port.call.up3, −1]:
  VAL INT [floors] port.goto IS [port.goto1, port.goto2, port.goto3, port.goto4]:
  VAL INT [floors] port.sensor IS [port.sensor1, port.sensor2, port.sensor3, port.sensor4]:
  BOOL obstruction.state:
  BOOL panic.state:
  WHILE TRUE
    TIME 20 MSEC −− Polling interval
    BOOL bit:
    SEQ
      SEQ i = 1 FOR floors−1
        SEQ
          Io.Read.Bit(handle, port.call.down[i], bit)
          IF
            bit AND (NOT call.down.state[i])
              input ! call ; i ; dir.down
            TRUE
              SKIP
          call.down.state[i] := bit
      SEQ i = 0 FOR floors−1
        SEQ
          Io.Read.Bit(handle, port.call.up[i], bit)
          IF
            bit AND (NOT call.up.state[i])
              input ! call ; i ; dir.up
            TRUE
              SKIP
          call.up.state[i] := bit
      SEQ i = 0 FOR floors
        SEQ
          Io.Read.Bit(handle, port.goto[i], bit)
          IF
            bit AND (NOT goto.state[i])
              input ! goto ; i
            TRUE
              SKIP
          goto.state[i] := bit
      SEQ i = 0 FOR floors
        SEQ
          Io.Read.Bit(handle, port.sensor[i], bit)
          IF
            bit AND (NOT sensor.state[i])
              input ! sensor ; i
            TRUE
              SKIP
          sensor.state[i] := bit
      SEQ
        Io.Read.Bit(handle, port.panic, bit)
        IF
          bit AND (NOT panic.state)
            input ! panic.button
          TRUE
            SKIP
        panic.state := bit
      SEQ
        Io.Read.Bit(handle, port.obstruction, bit)
        IF
          NOT bit = obstruction.state
            input ! obstruction ; bit
          TRUE
            SKIP
        obstruction.state := bit
:

PROC Io.Clear(VAL INT handle)
```

```
  VAL INT [15] dig.ports IS [port.light.goto1, port.light.goto2, port.light.goto3, port.light.goto4,
      port.light.down2, port.light.down3, port.light.down4, port.light.up1,
      port.light.up2, port.light.up3, port.floor.ind1, port.floor.ind2, port.motordir, port.panic.light,
      port.door.open]:
  SEQ i = 0 FOR SIZE dig.ports
    Io.Set.Bit(handle, dig.ports[i], FALSE)
:

PROC Io(CHAN IO.OUTPUT output, CHAN IO.INPUT input)
  INT handle:
  SEQ
    TIME 10 MSEC -- initialization
      SEQ
        Io.Init(handle)
        Io.Clear(handle)
    PAR
      Io.Output(handle, output)
      Io.Input(handle, input)
:
--}}}
-- *** IO SIGNAL DEMUX ***--{{{
PROTOCOL DEMUX.BUTTON
  CASE
    call ; FLOOR ; DIR
    goto ; FLOOR
:

PROTOCOL DEMUX.BUTTON.LIGHT
  CASE
    call ; FLOOR ; DIR ; BOOL
    goto ; FLOOR ; BOOL
:

INT FUNCTION get.speed(VAL DIR dir)
  INT speed:
  VALOF
    IF
      dir = dir.up
        speed := 500
      dir = dir.down
        speed := −500
      TRUE
        speed := 0
    RESULT speed
:

PROC Demux.Input(CHAN IO.INPUT input, CHAN DEMUX.BUTTON button,
    CHAN FLOOR sensor, CHAN BOOL obstruction, EVENT panic)
  WHILE TRUE
    ALT
      FLOOR f:
      DIR d:
      BOOL b:
      input ?? CASE
        call ; f ; d
          button ! call ; f ; d
        goto ; f
        button ! goto ; f
      panic.button
        RAISE panic
      sensor ; f
        sensor ! f
      obstruction ; b
        obstruction ! b
:

PROC Demux.Output(CHAN IO.OUTPUT output, CHAN DEMUX.BUTTON.LIGHT button.light,
    CHAN FLOOR indicator, CHAN BOOL door.light, panic.light, CHAN DIR motor)
  WHILE TRUE
    FLOOR f:
    BOOL b:
    DIR d:
    ALT
      button.light ?? CASE
        call ; f ; d ; b
          output ! call.light ; f ; d ; b
        goto ; f ; b
          output ! goto.light ; f ; b
      indicator ?? f
        output ! floor.indicator ; f
      door.light ?? b
        output ! door.light ; b
      panic.light ?? b
        output ! panic.light ; b
      motor ?? d
        output ! motor.speed ; get.speed(d)
:

PROC Demux(CHAN DEMUX.BUTTON button,
    CHAN FLOOR sensor, CHAN BOOL obstruction, EVENT panic,
    CHAN DEMUX.BUTTON.LIGHT button.light, CHAN FLOOR indicator,
    CHAN BOOL door.light, panic.light, CHAN DIR motor)
  CHAN IO.INPUT input:
  CHAN IO.OUTPUT output:
  PAR
    Io(output, input)
    Demux.Input(input, button, sensor, obstruction, panic)
    Demux.Output(output, button.light, indicator, door.light, panic.light, motor)

:

--}}}
-- *** INIT AND PANIC FILTER ***--{{{
PROC Filter.Input(CHAN DEMUX.BUTTON button.in, CHAN FLOOR sensor.in,
    CHAN BOOL obstruction.in, CHAN BOOL panic.in, CHAN INTERFACE.BUTTON button.out,
    CHAN FLOOR sensor.out, CHAN BOOL obstruction.out, EVENT no.panic)
  BOOL panic:
  BOOL obstruction.off:
  WHILE TRUE
    FLOOR f:
    DIR d:
    ALT
      NOT panic & button.in ?? CASE
        call ; f ; d
```

```
          button.out ! call ; f ; d
        goto ; f
          button.out ! goto ; f
      panic & button.in ?? CASE
        call ; f ; d
          SKIP
        goto ; f
          RAISE no.panic
      sensor.in ?? f
        sensor.out ! f
      BOOL b:
      obstruction.in ?? b
        IF
          NOT panic
            obstruction.out ! b
          NOT b
            obstruction.off := TRUE
          TRUE
            SKIP
      panic.in ?? panic
        IF
          NOT panic AND obstruction.off
            SEQ
              obstruction.out ! FALSE
              obstruction.off := FALSE
          TRUE
            SKIP
:

PROC Filter.Output(CHAN DEMUX.BUTTON.LIGHT button.light.out, CHAN FLOOR indicator.out,
    CHAN BOOL door.light.out, CHAN DIR motor.out, CHAN BOOL panic.in,
    CHAN INTERFACE.BUTTON.LIGHT button.light.in, CHAN FLOOR indicator.in,
    CHAN BOOL door.light.in, CHAN DIR motor.in, EVENT no.panic)
  BOOL panic:
  DIR last.motor:
  SEQ
    last.motor := dir.none
    WHILE TRUE
      ALT
        panic.in ?? panic
          IF
            panic = FALSE
              motor.out ! last.motor
            TRUE
              motor.out ! dir.none
        FLOOR f:
        DIR d:
        BOOL v:
        button.light.in ? CASE
          call ; f ; d ; v
            button.light.out ! call ; f ; d ; v
          goto ; f ; v
            button.light.out ! goto ; f ; v
        FLOOR f:
        indicator.in ? f
          indicator.out ! f
        BOOL v:
```

```
        door.light.in ? v
          door.light.out ! v
        motor.in ? last.motor
          IF
            NOT panic
              motor.out ! last.motor
            TRUE
              SKIP
:

PROC Update.Panic(BOOL panic.state, VAL BOOL new.state, CHAN [] BOOL panic.signal,
    CHAN BOOL panic.light, CHAN NULL clear.queue)
  IF
    NOT panic.state = new.state
      SEQ
        IF
          new.state = TRUE
            clear.queue ! null
          TRUE
            SKIP
        panic.state := new.state
        SEQ i = 0 FOR SIZE panic.signal
          panic.signal ! new.state
        panic.light ! new.state
:

PROC Filter.Panic(EVENT panic.event, no.panic.event, CHAN [] BOOL panic.signal,
    CHAN BOOL panic.light, CHAN NULL clear.queue)
  CHAN NULL do.panic.signal, dont.panic.signal:
  BOOL panic.state:
  PAR
    WHILE TRUE
      HANDLE panic.event
        TIME 10 MSEC -- panic event
          SEQ
            do.panic.signal ! null
    WHILE TRUE
      HANDLE no.panic.event
        TIME 1 SEC -- end-of-panic event
          SEQ
            dont.panic.signal ! null
    WHILE TRUE
      ALT
        do.panic.signal ?? CASE
          null
            Update.Panic(panic.state, TRUE, panic.signal, panic.light, clear.queue)
        dont.panic.signal ?? CASE
          null
            Update.Panic(panic.state, FALSE, panic.signal, panic.light, clear.queue)
:

PROC Filter(CHAN INTERFACE.BUTTON button, CHAN FLOOR sensor,
    CHAN BOOL obstruction, CHAN INTERFACE.BUTTON.LIGHT button.light, CHAN FLOOR indic
    CHAN BOOL door.light, CHAN DIR motor, CHAN NULL clear.queue)
  CHAN DEMUX.BUTTON demux.button:
  CHAN FLOOR demux.sensor:
  CHAN BOOL demux.obstruction:
```

```
  CHAN BOOL demux.door.light:
  CHAN DEMUX.BUTTON.LIGHT demux.button.light:
  CHAN FLOOR demux.indicator:
  CHAN BOOL demux.panic.light:
  CHAN DIR demux.motor:
  CHAN [2] BOOL panic.signal:
  EVENT demux.panic:
  EVENT no.panic:
  PAR
    Filter.Panic(demux.panic, no.panic, panic.signal, demux.panic.light, clear.queue)
    Filter.Input(demux.button, demux.sensor, demux.obstruction, panic.signal[0],
        button, sensor, obstruction, no.panic)
    Filter.Output(demux.button.light, demux.indicator, demux.door.light,
        demux.motor, panic.signal[1], button.light, indicator, door.light, motor, no.panic)
    Demux(demux.button, demux.sensor, demux.obstruction, demux.panic,
        demux.button.light, demux.indicator, demux.door.light,
        demux.panic.light, demux.motor)
:
--}}}
--}}}
-- *** QUEUE ***--{{{
PROTOCOL QUEUE.REQUEST
  CASE
    what.now ; FLOOR ; DIR
:

PROTOCOL QUEUE.REPLY
  CASE
    move ; DIR
    sleep
    open
:

INT FUNCTION queue.index(VAL FLOOR f, VAL DIR d)
  INT rv:
  VALOF
    IF
      d = dir.down
        rv := f
      d = dir.up
        rv := f + floors
      TRUE
        rv := 2 * f + floors
    RESULT rv
:

PROC Clear.Queue(BOOL [] queue, CHAN INTERFACE.BUTTON.LIGHT light)
  SEQ f = 0 FOR floors
    SEQ
      queue[queue.index(f, dir.down)] := FALSE
      queue[queue.index(f, dir.up)] := FALSE
      queue[queue.index(f, dir.none)] := FALSE
      light ! call ; f ; dir.down ; FALSE
      light ! call ; f ; dir.up ; FALSE
      light ! goto ; f ; FALSE
:
```

```
PROC Remove.Queued(BOOL [] queue, CHAN INTERFACE.BUTTON.LIGHT light,
    FLOOR floor, DIR dir)
  SEQ
    queue[queue.index(floor, dir)] := FALSE
    queue[queue.index(floor, dir.none)] := FALSE
    light ! call ; floor ; dir ; FALSE
    light ! goto ; floor ; FALSE
:

PROC Raise.Wake.Up(EVENT wake.up, BOOL sleeping)
  IF
    sleeping
      SEQ
        RAISE wake.up
        sleeping := FALSE
    TRUE
      SKIP
:

BOOL FUNCTION anybody.above(VAL BOOL [] queue, VAL FLOOR floor)
  BOOL rv:
  VALOF
    IF
      IF i = floor+1 FOR floors−floor−1
        IF d = 0 FOR dirs
          queue[queue.index(i, d)]
            rv := TRUE
      TRUE
        rv := FALSE
    RESULT rv

:

BOOL FUNCTION anybody.below(VAL BOOL [] queue, VAL FLOOR floor)
  BOOL rv:
  VALOF
    IF
      IF i = 0 FOR floor
        IF d = 0 FOR dirs
          queue[queue.index(i, d)]
            rv := TRUE
      TRUE
        rv := FALSE
    RESULT rv

:

BOOL FUNCTION anybody.here(VAL BOOL [] queue, VAL FLOOR floor, VAL DIR dir)
  VALOF
    SKIP
    RESULT queue[queue.index(floor, dir)] OR queue[queue.index(floor,dir.none)]

:

PROC What.to.do(CHAN QUEUE.REPLY reply, CHAN INTERFACE.BUTTON.LIGHT light,
    BOOL [] queue, FLOOR floor, DIR dir, BOOL sleeping)
  IF
```

```
    anybody.here(queue, floor, dir)
      SEQ
        Remove.Queued(queue, light, floor, dir)
        reply ! open
    dir = dir.up AND anybody.above(queue, floor)
      SEQ
        reply ! move ; dir.up
    dir = dir.down AND anybody.below(queue, floor)
      SEQ
        reply ! move ; dir.down
    -- NOTE: for completeness only: should never happen
    anybody.below(queue, floor)
      SEQ
        reply ! move ; dir.down
    anybody.above(queue, floor)
      SEQ
        reply ! move ; dir.up
    TRUE
      SEQ
        sleeping := TRUE
        reply ! sleep
:

PROC Queue(EVENT wake.up, CHAN INTERFACE.BUTTON button,
    CHAN INTERFACE.BUTTON.LIGHT button.light, CHAN QUEUE.REQUEST request,
    CHAN QUEUE.REPLY reply, CHAN NULL clear.queue, same.floor.button)
  BOOL [3*floors] queue:
  FLOOR current.floor:
  DIR current.dir:
  BOOL sleeping:
  WHILE TRUE
    ALT
      FLOOR f:
      DIR d:
      button ?? CASE
        call ; f ; d
          SEQ
            button.light ! call ; f ; d ; TRUE
            queue[queue.index(f, d)] := TRUE
            Raise.Wake.Up(wake.up, sleeping)
        goto ; f
          IF
            f = current.floor AND current.dir = dir.none
              SEQ
                same.floor.button ! null
            TRUE
              SEQ
                button.light ! goto ; f ; TRUE
                queue[queue.index(f, dir.none)] := TRUE
                Raise.Wake.Up(wake.up, sleeping)
      request ? CASE
        what.now ; current.floor ; current.dir
          SKIP
          SEQ
            What.to.do(reply, button.light, queue, current.floor, current.dir, sleeping)
      clear.queue ?? CASE
        null
```

```
          Clear.Queue(queue, button.light)
:

--}}}
-- *** MOVER ***--{{{
PROC Mover.Idle(EVENT wake.up, open.door, CHAN QUEUE.REQUEST request,
    CHAN QUEUE.REPLY reply, CHAN FLOOR floor, CHAN DIR motor, BOOL idle)
  HANDLE wake.up
    TIME 1 SEC -- wake up from idle
      FLOOR f:
      DIR d:
      SEQ
        floor ? f
        request ! what.now ; f ; dir.none
        reply ? CASE
          move ; d
            SEQ
              motor ! d
              idle := FALSE
          sleep
            SKIP
          open
            RAISE open.door
:

PROC Mover.Moving(EVENT sensor, open.door, CHAN QUEUE.REQUEST request,
    CHAN QUEUE.REPLY reply, CHAN FLOOR floor, CHAN DIR motor, BOOL idle)
  HANDLE sensor
    TIME 100 MSEC -- floor sensor event
      FLOOR f:
      DIR d:
      SEQ
        floor ? f
        request ! what.now ; f ; dir.none
        reply ? CASE
          move ; d
            SEQ
              motor ! d
          sleep
            SEQ
              motor ! dir.none
              idle := TRUE
          open
            SEQ
              motor ! dir.none
              RAISE open.door
:

PROC Mover.Floor.Server(EVENT sensor.event, CHAN FLOOR sensor, floor, indicator)
  FLOOR f:
  WHILE TRUE
    ALT
      floor ! f
        SKIP
      sensor ?? f
        SEQ
          indicator ! f
          RAISE sensor.event
:

PROC Mover(CHAN QUEUE.REQUEST queue.request, CHAN QUEUE.REPLY queue.reply,
    EVENT wake.up, open.door, CHAN FLOOR sensor, indicator, CHAN DIR motor)
  BOOL idle:
  EVENT sensor.event:
  CHAN FLOOR floor:
  SEQ
    TIME 10 MSEC
      motor ! dir.up
    PAR
      Mover.Floor.Server(sensor.event, sensor, floor, indicator)
      WHILE TRUE
        IF
          idle
            Mover.Idle(wake.up, open.door, queue.request, queue.reply, floor, motor, idle)
          TRUE
            Mover.Moving(sensor.event, open.door, queue.request, queue.reply, floor, motor, idle)
:

--}}}
-- *** DOOR ***--{{{
PROC If.Watch.Raise.Obstructed.Clear.Watch(EVENT obstructed, BOOL watch)
  IF
    watch
      SEQ
        RAISE obstructed
        watch := FALSE
    TRUE
      SKIP
:

PROC Door.Obstruction.Server(EVENT obstructed, CHAN BOOL obstruction,
    CHAN BOOL obstruction.watch, CHAN NULL same.floor.button)
  BOOL watch:
  BOOL obs:
  WHILE TRUE
    ALT
      same.floor.button ?? CASE
        null
          If.Watch.Raise.Obstructed.Clear.Watch(obstructed, watch)
      obstruction ?? obs
        IF
          obs
            If.Watch.Raise.Obstructed.Clear.Watch(obstructed, watch)
          TRUE
            SKIP
      obstruction.watch ?? watch
        IF
          obs
            If.Watch.Raise.Obstructed.Clear.Watch(obstructed, watch)
          TRUE
            SKIP


:
```

```
PROC Door.Door(EVENT open.door, wake.up, obstructed, CHAN BOOL obstruction.watch, door.light)
  WHILE TRUE
    HANDLE open.door
      BOOL close:
      SEQ
        TIME 1 SEC -- opening the door
          SEQ
            door.light ! TRUE
            obstruction.watch ! TRUE
        WHILE NOT close
          HANDLE
            obstructed -- obstruction event
              TIME 10 MSEC
                obstruction.watch ! TRUE
            TIMEOUT 10 SEC -- keeping the door open
              TIME 1 SEC -- closing the door
                SEQ
                  close := TRUE
                  obstruction.watch ! FALSE
                  door.light ! FALSE
                  RAISE wake.up
:

PROC Door(EVENT open.door, wake.up, CHAN BOOL obstruction, CHAN BOOL door.light,
    CHAN NULL same.floor.button)
  CHAN BOOL obstruction.watch:
  EVENT obstructed:
  PAR
    Door.Obstruction.Server(obstructed, obstruction, obstruction.watch, same.floor.button)
    Door.Door(open.door, wake.up, obstructed, obstruction.watch, door.light)
:

--}}}
PROC Main()--{{{
  CHAN INTERFACE.BUTTON button:
  CHAN FLOOR sensor, indicator:
  CHAN BOOL obstruction:
  CHAN INTERFACE.BUTTON.LIGHT button.light:
  CHAN BOOL door.light:
  CHAN DIR motor:
  CHAN NULL clear.queue, same.floor.button:
  CHAN QUEUE.REQUEST queue.request:
  CHAN QUEUE.REPLY queue.reply:
  EVENT open.door, wake.up:
  PAR
    Filter(button, sensor, obstruction, button.light, indicator, door.light, motor, clear.queue)
    Queue(wake.up, button, button.light, queue.request, queue.reply, clear.queue, same.floor.button)
    Door(open.door, wake.up, obstruction, door.light, same.floor.button)
    Mover(queue.request, queue.reply, wake.up, open.door, sensor, indicator, motor)
:
--}}}
```

Appendix C

Nomenclature for Schedulability Analysis Part

THIS CHAPTER lists variable names and notation used in the schedulability analysis part of this thesis.

Properties of Tasks and Jobs:

$A, B, C, \ldots$	Used to denote tasks.
T_A	Period of A, or minimum inter-arrival time (MIT) if A is a sporadic task.
P_A	Static priority of A under fixed priority scheduling (FPS).
D_A	Relative deadline of A.
R_A	Worst-case response time of A.
C_A	Worst-case execution time (WCET) of one job of A.
U_A	Utilization of A, or C_A/T_A.
$\widehat{\pi}(A)$	Preemption level of A.
λ_A	Density of A, or C_A/D_A.
$A :: i$	Task A is currently executing statement i.
$a, b, c, \ldots$	Used to denote jobs.
d_a	Absolute deadline of a.
r_a	Release time of a.
$\pi(a)$	Current (effective) scheduling priority of a.

Properties of the System:

m	Number of processors in system.
$\mathbb{T}$	Set of tasks in the system.
$\lvert\mathbb{T}\rvert$	Number of tasks in the system.
U_{sum}	The utilization of a system, or $\sum_{X \in \mathbb{T}} C_X / T_X$.
λ_{sum}	The density of a system, or $\sum_{X \in \mathbb{T}} C_X / D_X$.

Servers and Shared Resources

$\mathbb{S}$	Set of resources shared under mutual exclusion, or set of servers.
$\mathrm{shared}(A, B)$	Set of resources shared between tasks A and B.
$\mathrm{holds}\, A$	Set of servers currently held by task A.
$\mathcal{D}(\cdot)$	Demand of statement, block or task.
$\lceil \widehat{\pi}(S) \rceil$	Preemption level ceiling of S.
B_π	Maximum blocking caused by tasks with preemption level less than π on tasks with a level higher than or equal to π.

Client–Server System Model of Chapter 6

$\mathrm{Exec}\ r$	Abstract statement requiring r units of fully preemptible CPU.
$\mathrm{Call}\ S.c$	Abstract statement performing call of type c to server S.
$\mathbb{I}$	Set of all statements.
I_S	Request phase blocks for server S.
$I_{S.c}$	Reply phase blocks for call c to server S.
Σ_S	Signature of server S, ie, set of call types accepted by S.
Q_S	Worst-case request time of S, ie, longest time it takes to accept a call, given that it is not currently held by another task.
$P_{S.c}$	Worst-case reply time of call c to server S, ie, longest time it takes to complete a call from the time it is accepted.
$\widehat{P}_S$	Worst-case reply time of any call to server S.

The Computation Time Process Model of Chapter 7

$\mathbb{P}$	Set of computation time processs (CTPs).
$\mathbb{S}$	Set of schedules.
$\mathbf{1}$	Process requiring one unit of non-preemptible CPU before terminating.
$\mathbf{0}$	Process requiring no CPU before terminating.
$P\,;Q$	P and Q in sequence.
$P \| Q$	P and Q interleaved.
$C(P)$	Total amount of computation in P.
$\mathcal{L}(P)$	Length of P, ie, minimum response time.
$\mathcal{H}(P)$	Immediate height of P, ie, the number of processors that P can utilize for its first step.
$P \sqsupseteq Q$	Q is an upper bound of P.
$P \leq Q$	P is easier to schedule than P.
$\mathsf{step}(P, m)$	The set of possible outcomes of executing a single step for P with m processors.
$P \otimes s$	The set of possible outcomes of executing P on schedule s.

The Computation Function Model of Chapter 8

σ	Upper bound on skew (unfairness) of the intra-job scheduler.
v	The virtual time of a job.
L	The minimum completion time of a job.
$c(v)$	A computation function.
$c^{\star}(v, T)$	The T-period extension of $c(v)$.
$c_a(v) \sqsupseteq c_b(c)$	Computation function $c_b(v)$ is an upper bound to $c_a(v)$.
$\mathcal{D}_a(t_0, t_1)$	Demand of job a in the window $[t_0, t_1]$.
$\delta_a(t_0, t_1)$	Interference to job a in the window $[t_0, t_1]$.
$w(t)$	The number of processors in use by higher priority jobs at time t.

Sequence Notation

$\langle a, b, c\rangle$	a followed by b, followed by c.
$\langle\rangle$	Empty sequence
$s \frown t$	Concatenation of sequences s and t.

Appendix D

Acronyms

BNF	Backus–Naur form.
BSP	bulk synchronous parallelism.
CPU	central processing unit.
CSCP	client-server ceiling protocol.
CSIP	client-server inheritance protocol.
CSP	communicating sequential processes.
CTP	computation time process.
DMPO	deadline-monotonic priority ordering.
DPS	Distributed Programming System.
EDF	earliest deadline first.
EDF-H	EDF-hybrid.
FPS	fixed priority scheduling.
JLP	job-level parallelism.
LL	Liu-Layland.
MIT	minimum inter-arrival time.
MPCP	multiprocessor PCP.
MSRP	multiprocessor SRP.
OOP	object-oriented programming.
OS	operating system.
PCP	priority ceiling protocol.
PDC	processor demand criterion.
PEARL	Process and Experiment Automation Real-time Language.
PFAIR	proportional-fair.
PIP	priority inheritance protocol.
POSIX	Portable Operating System Interface for Unix.
P-PCP	parallel PCP.
QPA	quick processor-demand analysis.
RMPO	rate-monotonic priority ordering.
RTA	response-time analysis.
RTC	Real-Time C.

RTSJ	Real-time Specification for Java.
SRP	stack resource policy.
WCET	worst-case execution time.
WYSIWYG	what-you-see-is-what-you-get.

Appendix E

Glossary

absolute deadline
: The clock time when a process must have finished its computation. 17, 103

aliasing
: Simultaneously referring to the same variable by different names, eg, by having separate pointers to the same memory address. 17, 54

aperiodic task
: A task that is released at irregular intervals, usually as a response to some event. 28

arbitrary deadline
: A task set has arbitrary deadlines, if the deadline of a task may be less than, equal to, or greater than its period. 41, 105

atomic
: An operation is atomic if it either completes or does nothing. 38

background task
: A task that is not given real-time constraints, and is instead only intended to be executed when no real-time tasks are ready. A lazy scheduler effectively prevents the use of background tasks. 52, 61, 62

base time
: The base time of a time-construct is the absolute time which is considered its release, and from which its absolute deadline and termination time is computed. 39, 58, 64

binary semaphore
: A semaphore with a 0–1 counter. Excess signals are discarded. 36, 60, 226

blocked process
: A process that is unable to proceed until progress is made by another process. 18, 225, 227, 228

clock–delay race
: The problem of using clock time to compute a relative delay: By the time the relative delay function is called, the computed value of the delay will have been invalidated by the passage of time. 34, 35

cohesion
: The degree of which different modules corresponds to different functional responsibilities. 10, 79

computation function
: Under a fair intra-job scheduler; the number of processors that a job can utilize as a function of its virtual time. 164, 219

concurrent
: A system is concurrent if it has multiple threads of execution executing simultaneously. 2

constrained deadline
: A task set has constrained deadlines, if no task has a relative deadline greater than its period. 26, 105, 110, 118, 164

coupling
: The degree of which the correctness of a module relies on details of the implementation of other modules. 10, 79

critical section
: A section of code using some resource, which should not be preempted by other tasks referring to the same resource. 14

deferred server call
: A server call is deferred if it delays computation generated by the call until after the call has completed. 117, 138

demand
: The demand of a task in a window is the amount of computation required by that task in the window. 224

demand-bound function
: The maximum sum of task demands in any window as a function of window length. 105, 136

density
: The density of task X is denoted λ_X, and is defined as C_X/D_X. 103

Dhall's effect
: The existence of multiprocessor systems that are unschedulable under earliest deadline first (EDF), even if their utilization is close to 1. 108

discovery
: In Toc, the process of evaluating control structures to look for **TIME** constructs that can be reached without executing primitive processes. 63

drift
: A periodic task experiences drift, if its effective period is greater than its intended period due to execution time overheads. 34, 41, 226

during process
: The process to execute during an extended rendezvous. 54, 67, 74, 225

exact schedulability test
: A test that provides sufficient and necessary conditions for schedulability. 103

extended rendezvous
: A type of rendezvous where an extra process, the during-process, is executed after the communicating processes have rendezvoused, but before they are released. 67, 225

fail-fast
: A system which fails upon detecting a condition that may later lead to failure. 61

fair scheduler
: A scheduler is fair if it distributes processors equally to all processes, even when considering arbitrary small windows in time. A completely fair scheduler is an idealization that is not fully realizable in practice. 5, 102

global scheduling
: In a multiprocessor system using global scheduling, tasks may migrate between processors. 108

handle–timeout
: A language primitive for waiting on an event, or timing out after a maximum waiting time has elapsed. 36, 37, 51, 58, 93

implicit deadline
: A task set has implicit deadlines, if all tasks have a relative deadline equal to its period. 26, 32, 78, 101, 104, 105

independent tasks
: Tasks are independent if no task may be blocked by another. 104

intra-job scheduler
: The scheduler that assigns processors given to a job to the parallel branches of that job. 4, 102, 146

jitter
: Variation in response times. 22, 27

job
: An instance of a task. 103, 217, 227

laziness hypothesis
: That any part of a real-time system that cannot be given a meaningful deadline for its completion can be omitted entirely (hypothesis 4.1). 52, 57

lazy scheduler
: A scheduler is lazy if it does not execute processes without an associated deadline, even if the system is otherwise idle. 52, 60, 223

local drift
: A periodic task experiences local drift if it experiences drift, but the magnitude of drift does not accumulate over time. 34, 36

malleable
: A parallel job is malleable if it may use a dynamic number of processors. 112, 145

moldable
: A parallel job is moldable, if it uses a fixed number of processors, the number of which is decided run-time by the scheduler. 112

monitor
: A set of shared variables together with a set of functions that can operate safely on those variables. 14, 15, 227

multiprocessor anomaly
: A common name for the observation that jobs arriving at maximum rate does not constitute a worst-case scenario for schedulability on a multiprocessor system. 109

mutex
: A mechanism for ensuring mutual exclusive access to a resource, such as a binary semaphore. 14, 226

mutual exclusion synchronization
: The sharing of resources between processes by using mutexes. 101, 115

NP-hard
: A problem that cannot be solved in polynomial time (ie, quickly), but where a suggested solution can be verified in polynomial time. 108

object-oriented programming
: A programming paradigm where data is encapsulated and bundled together with methods that operate on the data. 16

occam
: An imperative, process-oriented programming language. 229

partial order
: An ordering of objects that is not a total order, ie, some pairs of objects may not be comparable. 122

partitioned scheduling
: A multiprocessor system where each task is permanently assigned to one processor. 108

period
: The MIT of jobs from a task. 103

periodic task
: A task is that is released at regular intervals. 103

pessimistic
: A test that provides only sufficient conditions for schedulability; ie, it may in some cases reject systems that are in fact schedulable. 103

preemption level
: A total order of tasks with respect preemption: A task with higher preemption level may or may not preempt a lower level task; but a lower level preemption task will never preempt a higher level task. 19, 32, 103, 124

primitive process
: A type of process in occam that will never contain a sub-process. For example, an assignment is a primitive process, but a conditional is not. 53

priority driven scheduler
: A real-time scheduler that assigns fixed priorities to jobs, but not necessarily to tasks. This class of schedulers include both FPS and EDF. 17, 103, 108, 123

priority inheritance
: A synchronization protocol that ensures that each process has at least as high priority as any process that is blocked waiting for it. 31

priority inversion
: A situation where a high priority task is blocked waiting for lower priority tasks due to resource sharing. 18, 44, 64, 106, 229

process-oriented programming
: A programming paradigm derived from communicating sequential processes (CSP), based on synchronous communication between self-contained parallel processes. 2, 16

protected object
: The Ada version of the monitor. 14, 15

race condition
: A concurrency error where program correctness depends unexpectedly on the particular interleaving of concurrent threads. 14, 15

reactive system
: A real-time system that has no reference to absolute clock times in its specification. 22, 23, 33

real-time system
: A system is real-time if failure to meet timing requirements is considered an error. 1

real-time task
: A top-level process in a real-time system with its own associated timing requirements. 52, 102, 225–227

relative deadline
: The time between some computation is allowed to begin execution, and the time when it must have completed. 17, 103

rendezvous
: A synchronization point between a set of processes, where all must have arrived at the rendezvous point before any process is released. 15, 225

reply phase
: A server is in its reply phase if it is currently serving a client. 119

request phase
: A server is in its request phase if it is not yet ready to serve clients. The request phase may for example contain initialization code, or cleanup or post-processing of a previous call. 119

response
: An action by a module or system detectable by its environment. In this thesis, for imperative programs: the completion of a designated statement or block. 23–26, 35

response time
: The response-time is the time from release to completion of a task or job. 39, 105, 217

rigid
: A parallel job is rigid, if it always uses the same, fixed number of processors. 112

safety
: A program component is safe if it only has explicit failure modes. 10

schedulability analysis
: To analyze a complete system executing on a given computing platform in order to determine whether it satisfies its temporal constraints. 23, 26, 101

scheduling overload
: A situation where the capacity of the system is insufficient to complete all processes within their deadlines. 19, 39, 61

semaphore
: A task locking mechanism. The semaphore contains a counter and two operations: “wait” and “signal”. Signaling increments the counter; waiting decrements the counter; any task attempting to decrement the counter when it is zero will be blocked until a signal. 14, 36, 59, 223

slack
: The difference between relative deadline and computation, ie, the maximum time a job can be denied execution before it will miss its deadline. 109

sporadic task
: An aperiodic task with a minimum inter-release time. 26, 28, 37, 103

stall
: A process has stalled if it is waiting for a process that is not ready to execute due to a delay constraint. 45, 59, 79, 116, 117, 119

stimulus
: An event or trigger requiring a response. In this thesis, for imperative programs: something that enables the execution of a statement or block. 23–26, 35

suspend–resume
: A low-level concurrency mechanism with two methods: suspend() to block execution of a task and a resume() to continue it. Suspend–resume is considered harmful, and have been removed from several concurrency libraries, including Java's. 36, 46, 59

sustainable schedulability test
: A schedulability test is sustainable if systems deemed to be schedulable will remain schedulable when reducing their computation or relaxing their temporal constraints. 103, 159

synchronization protocol
: A protocol used to control accesses to resources in order to avoid priority inversions and improve schedulability. 101, 103, 106, 227

task
: A top-level process. 102, 115, 217

temporal constraint
: A temporal constraint, or timing requirement, is a part of the specification of a real-time system that refers to time. 23, 25, 26, 30, 229

temporal scope
: A block of statements with associated temporal constraints. 24, 25, 33, 39

time-construct
: A language primitive that specifies an equal minimum and maximum execution time constraint to a block of code. 39, 41, 42, 45–49, 51, 52, 58, 64, 185, 187, 223

Toc
: Short for TIME/occam; an experimental programming language based on occam developed and implemented as part of this thesis. 4, 51

total order
: An ordering of objects so that for any pair of objects, one object is—in some sense—less than or equal to the other. 19, 152, 226, 227

Transputer
: An early microprocessor capable of parallel execution, intended to be used in clusters. 16

usage rule
: Rules in occam that prevent aliasing and illegal sharing of a variable between parallel processes. 54, 69

utilization
: The utilization of task X is denoted U_X, and is defined as C_X/T_X. 103, 224

virtual time
: Under a fair intra-job scheduler; the virtual time represents the progress of a job, measured by how long it will take for the job to reach its current state if always getting as many processors as it can utilize. 164, 219

work-conserving scheduler
: A scheduler that never leaves a processor idle if the system contains jobs that are ready to execute. 4, 146, 148

worst-case reply time
: The maximum amount of time required by server S from accepting a call of type $S.c$ to the corresponding reply. 126

worst-case request time
: The maximum amount of time required by a server before accepting some pending request, given that it is not serving other clients at the time. 126

References

[1] N. C. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8:284–292, 1993.

[2] T. P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, 1991.

[3] T. P. Baker. Multiprocessor EDF and deadline monotonic schedulability analysis. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, pages 120–129. IEEE Computer Society, 2003.

[4] T. P. Baker. Further improved schedulability analysis of EDF on multiprocessor platforms. Technical Report TR-051001, Dept. of Computer Science, Florida State University, Tallahassee, FL, USA, 2005.

[5] T. P. Baker and A. Shaw. The cyclic executive model and Ada. In *Proceedings of the 1988 Real-Time Systems Symposium*, pages 120–129, Dec. 1988.

[6] S. K. Baruah. Resource sharing in EDF-scheduled systems: a closer look. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 379–387. IEEE Computer Society, 2006.

[7] S. K. Baruah. Techniques for multiprocessor global schedulability analysis. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 119–128. IEEE Computer Society, 2007.

[8] S. K. Baruah and T. P. Baker. Schedulability analysis of global EDF. *Real-Time Systems*, 38(3):223–235, 2008.

[9] S. K. Baruah and A. Burns. Sustainable scheduling analysis. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 159–168. IEEE Computer Society, 2006.

[10] S. K. Baruah, R. R. Howell, and L. E. Rosier. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2(4):301–324, 1990.

[11] S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th IEEE International Real-Time Systems Symposium*, pages 182–190. IEEE Computer Society, 1990.

[12] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6): 600–625, 1996.

[13] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[14] V. Berten, S. Collette, and J. Goossens. *Feasibility Test for Multi-Phase Parallel Real-Time Jobs*, pages 33–36. IEEE Computer Society, 2009.

[15] M. Bertogna and M. Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 149–160. IEEE Computer Society, 2007.

[16] M. Bertogna, M. Cirinei, and G. Lipari. Improved schedulability analysis of EDF on multiprocessor platforms. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 209–218. IEEE Computer Society, 2005.

[17] M. Bertogna, M. Cirinei, and G. Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on Parallel and Distributed Systems*, 20(4):553–566, Apr. 2009.

[18] E. Bini, G. C. Buttazzo, and G. M. Buttazzo. Rate monotonic analysis: the hyperbolic bound. *IEEE Transactions on Computers*, 52(7):933–942, 2003.

[19] H.-J. Boehm. Threads cannot be implemented as a library. In *Proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation*, pages 261–268. ACM, 2005.

[20] P. Brinch Hansen. Structured multiprogramming. *Communications of the ACM*, 15(7), July 1972.

[21] P. Brinch Hansen. Java's insecure parallelism. *ACM SIGPLAN Notices*, 34 (4), Apr. 1999.

[22] F. P. Brooks. No silver bullet—Essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[23] N. C. C. Brown. Tock—beginning with Omega. Presentation, Computing Laboratory, University of Kent, Jan. 2008. www.cs.kent.ac.uk/archive/people/rpg/nccb2/.

[24] P. A. Buhr. Are safe concurrency libraries possible? *Communications of the ACM*, 38(2):117–120, Feb. 1995.

[25] R. E. Burkard and E. Çela. Linear assignment problems and extensions. In D. Du and P. M. Pardalos, editors, *Handbook of Combinatorial Optimization*, pages 75–149. Kluwer Academic Publishers, 1999. supplement volume A.

[26] A. Burns and A. J. Wellings. Synchronous sessions and fixed priority scheduling. *Journal of Systems Architecture*, 44(2):107–118, 1997.

[27] A. Burns and A. J. Wellings. *Concurrent And Real-Time Programming In Ada*. Cambridge University Press, Cambridge, UK, 2007.

[28] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages*. Pearson Education Limited, Essex, England, fourth edition, 2009.

[29] A. Burns, B. Dobbing, and T. Vardanega. Guide for the use of the Ada Ravenscar profile in high integrity systems. *ACM SIGAda Ada Letters*, 24 (2):1–74, 2004.

[30] A. Burns, A. J. Wellings, and T. Taft. Supporting deadlines and EDF scheduling in Ada. In A. Llamosí and A. Strohmeier, editors, *Proceedings of the Ada Europe Conference on Reliable Software Technologies*, pages 156–165. Springer-Verlag Berling Heidelberg, 2004.

[31] G. C. Buttazzo. Rate monotonic vs. EDF: Judgment day. *Real-Time Systems*, 29(1):5–26, 2005.

[32] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Conference Record of the 14th Annual ACM Symposium on the Principles of Programming Languages*, 1987.

[33] A. Cervin, J. Eker, B. Bernhardsson, and K.-E. Årzén. Feedback-feedforward scheduling of control tasks. *Real-Time Systems*, 23, 2002.

[34] S. Collette, L. Cucu, and J. Goossens. Integrating job parallelism in real-time scheduling theory. *Information Processing Letters*, 106(5):180–187, May 2008.

[35] D. Cornhilll, L. Sha, and J. P. Lehoczky. Limitations of Ada for real-time scheduling. *ACM SIGAda Letters*, VII(6):33–39, 1987.

[36] P. Crescenzi and V. K. (eds.). A compendium of NP optimization problems. www.nada.kth.se/~viggo/problemlist/, 2000.

[37] B. Dasarathy. Timing constraints of real-time systems: Constructs for expressing them, methods of validating them. *IEEE Transactions on Software Engineering*, SE–11(1):80 – 86, Jan. 1985.

[38] U. C. Devi, H. Leontyev, and J. H. Anderson. Efficient synchronization under global EDF scheduling on multiprocessors. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 75–84. IEEE Computer Society, 2006.

[39] E. W. Dijkstra. Notes on structured programming. In *Structured programming*, pages 1–82. Academic Press Ltd., London, UK, 1972.

[40] A. Easwaran and B. Anderson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. Technical report, Polytechnic Institute of Porto, 2009.

[41] J. L. L. et. al. Ariane 5: Flight 501 failure: Report by the inquiry board. Technical report, ESA, July 1996.

[42] C. J. Fidge. A formal definition of priority in CSP. *ACM Trans. Program. Lang. Syst.*, 15(4):681–705, 1993.

[43] *Failures-Divergence Refinement User Manual.* Formal Systems (Europe) Ltd., June 2005. Web page: www.fsel.com.

[44] P. Gai, M. D. Natale, G. Lipari, A. Ferrari, G. Gabellini, and P. Marceca. A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 189–198. IEEE Computer Society, 2003.

[45] N. Gehani and K. Ramamritham. Real-time concurrent C: A language for programming dynamic real-time systems. *Real-Time Systems*, 3(4):377–405, 1991.

[46] *PEARL 90 Language Report.* GI-Working Group 4.4.2, 1998.

[47] J. Goossens and V. Berten. Gang FTP scheduling of periodic and parallel rigid real-time tasks. In *18th International Conference on Real-Time and Network Systems*, 2010.

[48] J. Goossens, S. Funk, and S. K. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, 25(2-3):187–205, Sept. 2003.

[49] K. N. Gregertsen and A. Skavhaug. Implementing the new Ada 2005 timing event and execution time control features on the AVR32 architecture. *Journal of Systems Architecture*, 56:509–522, 2010.

[50] N. Halbwachs. A synchronous language at work: the story of LUSTRE. In *Proceedings from the Third ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, pages 3–11, July 2005.

[51] C.-C. Han and K.-J. Lin. Scheduling parallelizable jobs on multiprocessors. In *Proceedings of the 1989 IEEE Real-Time Systems Symposium*, pages 59–67, 1989.

[52] L. Hatton. Safer language subsets: an overview and a case history, MISRA C. *Information and Software Technology*, 46(7):465–472, 2004.

[53] M. Herlihy. A methodology for implementing highly concurrent data objects. *Transactions on Programming Languages and Systems*, 15:745–770, 1993.

[54] C. A. R. Hoare. Hints on programming language design. Technical report, Stanford University, Stanford, CA, USA, 1973.

[55] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10), Oct. 1974.

[56] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1978.

[57] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison-Wesley Professional, 2003.

[58] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[59] S. Kato and Y. Ishikawa. Gang EDF scheduling of parallel task systems. In *Proceedings of the IEEE International Real-Time Systems Symposium 2009*, pages 459–468. IEEE Computer Society, 2009.

[60] E. Kligerman and A. D. Stoyenko. Real-time euclid: A language for reliable real-time systems. *IEEE Transactions of Software Engineering*, pages 941–949, 1986.

[61] M. Korsgaard and S. Hendseth. Combining EDF scheduling with occam using the Toc programming language. In A. A. McEwan, W. Ifill, and P. H. Welch, editors, *Communicating Process Architectures 2008*, pages 339–348. IOS Press, Sept. 2008.

[62] M. Korsgaard and S. Hendseth. Design patterns for communicating systems with deadline propagation. In P. H. W. et. al, editor, *Communicating Process Architectures 2009*, pages 349–361. IOS Press, Nov. 2009.

[63] M. Korsgaard and S. Hendseth. The computation time process model. In P. H. Welch, A. T. Sampson, J. B. Pedersen, J. Kerridge, J. F. Broenink, and F. R. M. Barnes, editors, *Communicating Process Architectures 2011*, pages 273–286. IOS Press, June 2011.

[64] M. Korsgaard and S. Hendseth. Schedulability analysis of malleable tasks with arbitrary parallel structure. In *Proceedings of the 17th International Conference on Real-Time Computing Systems and Applications*, pages 3–14. IEEE Computer Society, Aug. 2011.

[65] M. Korsgaard, A. Skavhaug, and S. Hendseth. Improving real-time software quality by direct specification of timing requirements. In *Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications*, pages 528–536. IEEE Computer Society, 2009.

[66] K. Lakshmanan, S. Kato, and R. R. Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Proceedings of the 2010 IEEE International Real-Time Systems Symposium*, pages 259–268. IEEE Computer Society, 2010.

[67] B. W. Lampson and D. L. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):105–117, Feb. 1980.

[68] A. E. Lawrence. CSPP and event priority. In A. G. Chalmers, M. Mirmehdi, and H. Muller, editors, *Communicating Process Architectures 2001*, pages 67–92, Sept. 2001.

[69] A. E. Lawrence. Overtures and hesitant offers: hiding in CSPP. In J. F. Broenink and G. H. Hilderink, editors, *Communicating Process Architectures 2003*, pages 97–109, Sept. 2003.

[70] I. Lee, V. Gehlot, L. Lee, V. Gehlot, I. Lee, and V. Gehlot. Language constructs for distributed real-time programming. In *Proceedings of the 1985 IEEE Real-Time Systems Symposium*, Dec. 1985.

[71] J. Y. T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982.

[72] N. Leveson. *Safeware: System Safety and Computers*, chapter Medical Devices: The Therac-25. Addison-Wesley, 1995.

[73] B. Li, B. Xu, and H. Yu. Transforming Ada serving tasks into protected objects. In *Proceedings of the 1998 annual ACM SIGAda International Conference on Ada*, pages 240–245. ACM, 1998.

[74] A. Lister. The problem of nested monitor calls. *Operating Systems Review*, 11(3):5, 1977.

[75] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[76] G. Lowe. Implementing generalised Alt. In P. H. Welch, A. T. Sampson, J. B. Pedersen, J. Kerridge, J. F. Broenink, and F. R. M. Barnes, editors, *Communicating Process Architectures 2011*, pages 1–34, jun 2011.

[77] S. Marlow. *The Glasgow Haskell compiler*, retrieved 30. Nov 2011. www.haskell.org/ghc/.

[78] J. M. R. Martin and P. H. Welch. A design strategy for deadlock-free concurrent systems. *Transputer Communications*, 3(4), June 1997.

[79] S. McConnell. *Code Complete: A practical handbook of software construction.* Microsoft Press, Redmond, Washington, second edition, 2004.

[80] M. M. Michael. Scalable lock-free dynamic memory allocation. *SIGPLAN Notices*, 39(6):35–46, 2004.

[81] M. Pellauer, M. Forsberg, and A. Ranta. BNF converter: Multilingual front-end generation from labelled BNF grammars. Technical report, 2004.

[82] K. Poulsen. Tracking the blackout bug. *SecurityFocus*, Apr. 2004. www.securityfocus.com/news/8412.

[83] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:4–13, 1992.

[84] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proceedings of the 1988 Real-time Systems Symposium*, pages 259–269, Dec. 2003.

[85] *The real-time specification for Java.* The Real-Time for Java Expert Group, 1.0.2 edition, 2006. www.rtsj.org.

[86] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. In L. Kott, editor, *Automata, Languages and Programming*, volume 226 of *Lecture Notes in Computer Science*, pages 314–323. Springer Berlin / Heidelberg, 1986.

[87] I. Ripoll, A. Crespo, and A. K. Mok. Improvement in feasibility testing for real-time tasks. *Real-Time Systems*, 11(1):19–39, 1996.

[88] A. W. Roscoe and C. A. R. Hoare. The laws of occam programming. *Theoretical Computer Science*, 60(2):177 – 229, 1988.

[89] M. Saksena and P. Karvelas. Designing for schedulability: integrating schedulability analysis with object-oriented design. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, pages 101–108. IEEE Computer Society, 2000.

[90] D. Schleef. Linux control and measurement device interface (comedi). www.comedi.org.

[91] *occam® 2.1 Reference Manual.* SGS-THOMPSON Microelectronics Limited, 1995. wotug.org/occam/documentation/oc21refman.pdf.

[92] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39 (9):1175–1185, Sept. 1990.

[93] D. B. Skillicorn, J. M. D. Hill, and W. F. Mccoll. Questions and answers about BSP, Nov. 1996. Oxford University Computing Laboratory.

[94] M. Spuri. Analysis of deadline scheduled real-time systems. Technical Report 2772, Institut National de Recherche en Informatique et en Automatique, Jan. 1996.

[95] A. Srinivasan. *Efficient and flexible fair scheduling of real-time tasks on multiprocessors*. PhD thesis, The University of North Carolina at Chapel Hill, 2003.

[96] W. P. Stevens, G. Myers, and L. Constantine. Structured design. In *Classics in software engineering*. Yourdon Press, Upper Saddle River, NJ, USA, 1979.

[97] *Threads and Swing*. Sun Developer Network, 2000. java.sun.com/products/jfc/tsc/articles/threads/threads1.html.

[98] H. Sundell. *Efficient and Practical Non-Blocking Data Structures*. PhD thesis, Chalmers University of Technology and Göteborg University, 2004.

[99] Ø. Teig. Safer multitasking with Communicating Sequential Processes (CSP). *Embedded Systems*, pages 57–68, June 2000.

[100] H. Thimbleby. A critique of Java. *Software — Practice and Experience*, 29 (5), 1999.

[101] M. Törngren. Fundamentals of implementing real-time control applications in distributed computer systems. *Real-Time Systems*, 14:219–250, 1998.

[102] P. H. Welch. Process oriented design for Java: Concurrency for all. volume 2330 of *Lecture Notes in Computer Science*, pages 687–687. Springer-Verlag, Apr. 2002.

[103] P. H. Welch. *An occam-Pi Quick Reference*. Computing Laboratory, University of Kent, retrieved 27. Oct 2011. www.cs.kent.ac.uk/research/groups/plas/wiki/OccamPiReference/.

[104] P. H. Welch and F. R. M. Barnes. Communicating mobile processes: introducing occam-pi. In A. E. Abdallah, C. B. Jones, and J. W. Sanders, editors, *25 Years of CSP*, volume 3525 of *Lecture Notes in Computer Science*, pages 175–210. Springer-Verlag, Apr. 2005.

[105] P. H. Welch, G. Justo, and C. Willcock. High-level paradigms for deadlock-free high-performance systems. In *Proceedings of the 1993 World Transputer Congress on Transputer Applications and Systems*, pages 981–1004. IOS Press, 1993.

[106] P. H. Welch, F. R. M. Barnes, and F. A. C. Polack. Communicating complex systems. In M. G. Hinchey, editor, *Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems*, 2006.

[107] V. Yodaiken. Against priority inheritance. Technical report, FSMLabs, 2002.

[108] D. Q. Zhang, C. Cecati, and E. Chiricozzi. Some practical issues of the transputer based real-time systems. In *Proceedings of the 1992 International Conference on Industrial Electronics, Control, Instrumentation and Automation*, pages 1403–1407. IEEE CNF, 1992.

[109] F. Zhang and A. Burns. Schedulability analysis for real-time systems with EDF scheduling. *IEEE Transactions on Computers*, 58:1250–1258, 2009.